AD-A116 780 MARINE CORPS TACTICAL SYSTEMS SUPPORT ACTIVITY CAMP --ETC F/G 17/2
DIGITAL COMMUNICATIONS TERMINAL HIGH ORDER PROGRAMMING LANGUAGE--ETC(U)
NOV 80 K C SHUMATE, R E SAUER, G E ANDERSON
UNCLASSIFIED 24E005/U-TN-01-VOL-1 NL

END
DATE FILMED
09-82
DTIC

TECHNICAL NOTE 24E005/U-TN-01

(3)

AD A118780

DIGITAL COMMUNICATIONS TERMINAL HIGH ORDER PROGRAMMING LANGUAGE STUDY

(DCT HOL STUDY)
VOLUME ONE
26 NOVEMBER 1980

MARINE CORPS TACTICAL SYSTEMS SUPPORT ACTIVITY

CAMP PENDLETON CALIFORNIA 92055

Approved for public release; distribution unlimited.

82 08 31 018

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: TECHNICAL NOTE 24E005/U-TN-01	2. GOVT ACCESSION NO.: AD-A118780
3. RECIPIENT'S CATALOG NUMBER:	
4. TITLE (and Subtitle): DIGITAL COMMUNICATIONS TERMINAL HIGH ORDER PROGRAMMING LANGUAGE STUDY (DCT HOL STUDY) VOLUME ONE	5. TYPE OF REPORT & PERIOD COVERED: Final report, Vol. 1
	6. PERFORMING ORG. REPORT NUMBER:
7. AUTHOR(s): Maj K.C. SHUMATE USMC; Mr. R.E. SAUER; Maj G.E. ANDERSON USMC; Maj J.W. HOOPER USMC; Capt B.F. BRADY USMC	8. CONTRACT OR GRANT NUMBER(s):
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Marine Corps Tactical Systems Support Activity, Marine Corps Base, Camp Pendleton, CA 92055	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: DCT Project
11. CONTROLLING OFFICE NAME AND ADDRESS: Marine Corps Development and Education Command, Quantico, VA 22134	12. REPORT DATE: 26 November 1980
	13. NUMBER OF PAGES: 173
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office): Marine Corps Tactical Systems Support Activity, Marine Corps Base, Camp Pendleton, CA 92055	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT (of this Report)

Approved for Public Release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Approved for Public Release; distribution unlimited

18. SUPPLEMENTARY NOTES

See also Volume 2.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

HIGH LEVEL LANGUAGE
DIGITAL COMMUNICATIONS TERMINAL (DCT)
HIGH ORDER PROGRAMMING LANGUAGE
NSC800 MICROPROCESSOR
C PROGRAMMING LANGUAGE
COMPUTER PROGRAMS
DCT HOL STUDY
MICROPROCESSORS
DECISION MAKING
DELPHI

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This paper reports the results of a study to select a high order programming language for the development of computer programs for the digital communications terminal. All languages suitable for use with the NSC800 microprocessor were considered. The nine final candidates were evaluated by a methodology including benchmarking and determination of a figure of merit. During the conduct of the study it became clear that the program support environment must include both a minicomputer software engineering host, and a microcomputer development

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. ABSTRACT

system. The language selected is Interactive Systems "C." The system includes a cross-compiler running on a PDP-11 and generating code for the NSC800.

Technical Note #24E005/U-TN-01

Digital Communications Terminal High Order Programming Language Study (DCT HOL Study)

Prepared by:
Maj K. C. SHUMATE USMC
Mr. R. E. SAUER
Maj G. E. ANDERSON USMC
Maj J. W. HOOPER USMC
Capt B. F. BRADY USMC

Reviewed by:
LtCol J. J. O'MEARA USMC
LtCol D. R. LITTLE USMC

ABSTRACT: This paper reports the results of a study to select a high order programming language for the development of computer programs for the digital commmunications terminal. All languages suitable for use with the NSC800 microprocessor were considered. The nine final candidates were evaluated by a methodology including benchmarking and determination of a figure of merit. During the conduct of the study it became clear that the program support environment must include both a minicomputer software engineering host, and a microcomputer development system. The language selected is Interactive Systems "C." The system includes a cross-compiler running on a PDP-11 and generating code for the NSC800.

26 November 1980

Digital Communications Terminal
Tactical Systems Concepts and Requirements Branch
Marine Corps Tactical Systems Support Activity
Camp Pendleton, California

Table of Contents

Volume One

EXECUTIVE SUMMARY

Volume Two

APPENDICES

DCT HOL Study: Executive Summary

This paper summarizes the results of a study to specify a high order programming language (HOL) for the Digital Communications Terminal (DCT). The reader is assumed to be familiar with the DCT and the Marine Corps systems acquisition process. The study was tasked to MCTSSA by CG MCDEC 152043Z Jul 80 to be completed by 1 Dec 1980.

Critical issues related to the study are:

- Identification of a programming support environment (PSE), support hardware and software, related to the selected language.
- Use of benchmarks to obtain comparable efficiency statistics.
- Evaluation of mature languages (compiler available).
- Consideration that DCT programs must be easy to modify and expand.

The methodology of the study called for identification of a list of candidate languages, coding and execution of a benchmark program, and development of a figure of merit based upon 18 technical features of HOLs. Technical features include how a language represents data structures, controls flow of execution, and accesses assembly language. The features also include efficiency figures and consideration of the PSE. The benchmark constitutes a test of the specific compiler, while the figure of merit also measures the desirability of the language itself. Non-quantifiable factors were taken into account during a final period of integration of results.

All major programming languages and variants were potential candidates for examination by benchmark. Many languages, including Ada, CMS-2, and Jovial, were eliminated because no appropriate compiler existed. Many languages were eliminated as being only minor variations of widely used languages such as

UCSD PASCAL or Microsoft FORTRAN-80. Other languages, such as APL, BASIC, COBOL, LISP, MDL, Pilot and others, were eliminated by a preliminary analysis as being unsuitable for real-time applications. After the preliminary analysis, 22 languages were left for consideration. More detailed analysis, combined with execution of benchmarks, quickly eliminated 13 of them as being too slow, or otherwise poor in performance. The 9 remaining languages were: "C" (two versions), FORTRAN, PASCAL (two versions), PLI-80, PLMX, PLZ, and RATFOR. After execution of benchmarks and scoring the languages, the top 4 languages were the versions of "C" and PASCAL. The language selected by consensus of the principal analysts involved is the version of "C" provided by Interactive Systems. It uses a cross-compiler hosted on a PDP-11 to generate code for a microprocessor. "C" was the most efficient and received the highest score, 797 out of 1000 points, as a figure of merit. FORTRAN, the only DOD Inst 5000.31 language among the finalists, was the least efficient and received the lowest figure of merit, 522, of the final 9 languages. It took 4 times as long and nearly 3 times the memory of the selected language.

The program support environment was identified in a general sense. The PSE must include 3 levels of capability:

- A minicomputer software engineering host with sophisticated software tools.

- A microcomputer development system with in-circuit-emulation and logic analysis capabilities.

- The target system, an NSC800 in the DCT.

The decision as to a specific suite of PSE hardware and software should be deferred and considered as part of the decision regarding the DCT software development and maintenance strategy.

In summary, the DCT software should be developed in Interactive Systems "C" in a sophisticated program support environment.

SECTION 1. INTRODUCTION

1.1 Purpose

This paper reports the results of a study to determine the appropriate high order programming language (HOL) for software development for the U.S. Marine Corps Digital Communications Terminal (DCT).

1.2 Background

1.2.1 Reader Assumptions

It is assumed that the reader is familiar with the DCT program and the Marine Corps systems acquisition process.

1.2.2 Tasking

MCTSSA was tasked to do this study by CG MCDEC 152043Z Jul 80 which directed:

- Conduct DCT HOL Study
- Complete NLT 1 Dec 1980

1.2.3 Study Objective

The objective of the study is to select a high order programming language for DCT use.

1.2.4 History

The DCT is based upon the Litton Interactive Display Terminal (IDT), uses the RCA 1802 processor, and is limited to 32K bytes of program memory. Due to the nature of the RCA 1802 and the limited memory available, the IDT and

initial test versions of the DCT were programmed in assembly language. With advances in technology, it became possible to use a more capable microprocessor and include more memory in the DCT. The production DCT, the one being considered by this study, is to use the NSC800 microprocessor and have 128K bytes of memory. MCTSSA has been assigned responsibility for DCT program development and maintenance. As of the present time the precise software development strategy has not yet been specified. Development could be done by Litton, by MCTSSA on-site, or by some other contractor. The computer programs will be developed in a HOL rather than assembly language, hence this study. In June 1980, MCTSSA prepared a proposed Statement of Work (SOW) for the conduct of the study. MCTSSA and C^3 Division collaborated in the production of a final draft SOW. That draft SOW contained most of the technical language features for the figure of merit analysis, and has served as general guidance for the conduct of the study.

1.2.5 Three Microprocessors

This paragraph clarifies the relationships among the three microprocessors mentioned in this study.

1.2.5.1 RCA 1802 (COSMAC)

The RCA 1802 is a low-power (CMOS technology) microprocessor. It is slow and has a limited instruction set and unusual instruction set architecture. It was a good selection several years ago, but is an outdated device unsuitable for development of a series of large programs.

1.2.5.2 Zilog Z80

The Z80 is among the most capable and widely used 8-bit microprocessors. Over 300,000 current computer systems are based upon the Z80. Many compilers target the Z80 for various HOLs. It is a high-power-consumption (NMOS technology) device. It was used to execute the DCT benchmark programs.

1.2.5.3 NSC800

The NSC800, recently announced but not yet in production, has been specified (by the DCT HOL Study draft SOW) as the microprocessor to be used in the production DCT. It is a low-power implementation of the Z80 instruction set. It is advertised as being capable of operating at the same speed as the Z80, and of being in some ways architecturally superior. Section 5 discusses the NSC800 in greater detail. Since the NSC800 is not now available in a complete computer system, the DCT benchmarks were executed on the Z80.

1.3 Scope

The study was oriented toward currently identified DCT applications and based on the assumption that the DCT will use the NSC800 microprocessor. It did not address issues related to the desirability of a language for other Marine Corps applications and did not consider assembly language as a candidate for DCT program development. Critical issues related to the study are described below.

1.3.1 Program Support Environment (PSE)

The PSE is the environment within which the programmer creates, tests, debugs, and integrates the DCT software. It is critical to the cost of program development, the accuracy and operational effectiveness of the software, and to the effectiveness of life cycle support.

1.3.2 Benchmarks

It is essential that actual programs, similar to the DCT application, be prepared and run on a DCT-like processor. This "benchmarking" will give comparative figures upon which to base a language decision.

1.3.3 Language and Compiler

Each generic language, FORTRAN, COBOL, etc., will have different compiler implementations by different vendors. The benchmark numerical results of execution time and memory use are primarily a test of the specific compiler implementation. It is also important to evaluate the language itself in a static sense. That is, the appropriateness for DCT applications of the language data and control constructs and other features. Such features must be subjectively evaluated. Also included in the subjective evaluation are other vendor-specific characteristics such as access to bit-level manipulation, access to assembly language, and level of documentation.

1.3.4 Modifiability

Due to the nature of the DCT and its intended application, it is essential that DCT programs be easy to modify and expand.

1.4 Organization of the Report

Section 1 contains introductory material and provides a summary of conclusions. Section 2 is a summary of the technical approach applied to the languages listed in Section 3. Sections 4 and 5 describe language and processor characteristics basic to the conduct of the study, while Sections 6 through 8 contain the results of the study along with greater detail on the methodology. Section 9 integrates the results of the analysis and does some final comparisons. Section 10 presents conclusions. Appendix A contains references. Appendix B contains the code for the programs that were prepared and run to obtain benchmark results and to serve as examples for the language feature analysis. Appendices C, D, and E contain summary figures resulting from the benchmark and figure of merit analyses.

1.5 Summary of Conclusions

1.5.1 Selection of Language

The language selected for programming the DCT is Interactive Systems C. The compiler runs on a PDP-11 host and produces code for a Z80.

1.5.2 Program Support Environment

A second conclusion of the study is that the program support environment must include a minicomputer software engineering host, as well as a microcomputer development system.

SECTION 2. METHODOLOGY

This section describes in general terms the technical approach of the study. Additional detail is provided in Sections 5 through 8.

2.1 General. The methodology called for three steps in the conduct of the study:

- Benchmarks
- Figure of Merit
- Integration

These three steps are described below.

2.2 Benchmarks

A benchmark program was prepared that was based upon DCT-like applications, and also encompassed important features related to data and control structures, procedure invocation, passing of parameters, and arithmetic, including both positive and negative integers. The benchmark was implemented in each of the final candidate languages and was executed on a Z80. The most important results were the figures on execution time and program size. In addition to the objective benchmark results, the benchmark was important in providing an example of code in each language and in providing a subjective assessment of the quality of the PSE for each language.

2.3 Figure of Merit

In order to provide a measure of the desirability of each language, including both the language itself and its implementation by a specific vendor, a figure of merit (FOM) was derived for each of the final candidates. The figure of merit was derived by a linear combination of the product of weighted technical features of languages, times scores (from 0 to 1.0)

assigned by evaluators. More detail, and results, are provided in Sections 4 and 7. This process resulted in a single number, the figure of merit, which is a measure of the relative desirability of the language for DCT applications.

2.4 Integration

Neither the benchmark process nor the development of a figure of merit can address all issues related to language development. In fact, there are potential methodological problems in the application of the figure of merit. The very early definition of the project technical approach specified that neither the most efficient language nor the language with the highest FOM would necessarily be the one selected for DCT use. In order to ensure all issues are considered, include new information available after the FOM analysis, assess non-quantifiable factors, account for FOM weaknessess and variability, and break possible FOM ties, a separate integration step was included. The integration step consisted of a series of discussions among the principal analysts, and a final decision meeting with the project leader and the cognizant Branch Chief.

2.5 Study Participants

The study was conducted by MCTSSA personnel, with contractor effort to provide the benchmark facilities, a programming consultant, and general support. Ten MCTSSA personnel, military and civil service, took part in the process of assigning weights to language features, and nine MCTSSA and two NOSC personnel took part in the assignment of scores to languages. Most of the participants had master's degrees, primarily in computer science. The principal analysts were familiar with language and microprocessor issues, and with DCT applications.

SECTION 3. CANDIDATE LANGUAGES

This section describes the process of language selection and elimination, notes a number of languages which were serious contenders, and lists the nine final candidates which were evaluated.

3.1 General Criteria

In addition to the principal analysts' knowledge of a number of obvious candidates, the journals listed in Appendix A were searched for language surveys, advertisements, and discussions of language characteristics. Literally hundreds of languages (i.e., language/vendor combinations) were under preliminary consideration. The three major criteria for elimination from consideration are described below.

3.1.1 No Compiler

If no compiler existed which had the Z80, and hence the NSC800, as a target processor, the language could not be a candidate. This criteria eliminated a number of languages, viz. Ada, CMS-2, SPL/1, Jovial, and Tacpol, which would otherwise have been primary candidates.

3.1.2 Minor Variants

Minor variants of well-known languages were eliminated. For example, Microsoft FORTRAN-80 is a widely used microprocessor implementation of the ANSI-66 specification. Vendors who indicated the compiler was a Microsoft variation or similar ANSI-66 implementation were eliminated. Similarly, minor compiler-interpreter implementations of UCSD PASCAL, now licensed by SofTech, were eliminated.

3.1.3 Lack of Suitability or Use

A number of languages, including APL, BASIC, COBOL, LISP, Pilot, MDL, and others, were eliminated either because they served only a very small community of users, or were unsuitable for the application. Languages were eliminated

for having a combination of: poor data and control structures, lack of control over scope of definition of variables, poor modularity, lack of dynamic data structures, lack of access to assembly language, and a general reputation for not being a systems programming language. Certain implementations, e.g. Microsoft COBOL, overcame some of the limitations but at the cost of allowing such deviation from specifications (such as ANSI 74 COBOL) that they could hardly be called the same language. Some of the issues are addressed in the literature, for example reference (h), "A Survey of Microprocessor Languages," notes specifically that APL, COBOL, and FORTRAN are not systems programming languages. In addition to literature review and general discussion, a preliminary FOM was calculated for generic (non-vendor-specific) languages. The results are in Appendix E. APL, BASIC, and COBOL were eliminated partly on this basis. Despite its weak showing, FORTRAN was not eliminated. If it did best on the quantitative aspects of the benchmark it might be an attractive candidate since it is a DOD approved HOL.

3.1.4 Candidate Languages

After the elimination process described, there were 22 languages in 6 groups left to consider. They are listed, with the vendor providing them, in the paragraphs below.

3.2 "C"

- BDS "C"	Lifeboat Assoc.
- "C"	Interactive Systems Corp.
- "C"	Whitesmith, Ltd.

3.3 FORTRAN

- FORTRAN-IV	Cromemco
- FORTRAN-66	Zilog, Inc.
- FORTRAN-77	SofTech
- FORTRAN-80	Microsoft
- FORTRAN-80	Intel Corp.

3.4 FORTRAN Pre-Processors

- RATFOR	Cromemco
- RATFOR	The Software Works

3.5 FORTH and Derivatives

- FORTH	FORTH, Inc.
- STOIC II	Avocet Systems, Inc.

3.6 PASCAL

- ISO PASCAL	Whitesmith, Ltd.
- PASCAL 64000	Hewlett-Packard
- PASCAL/M	Digital Marketing
- PASCAL/MT	MT MicroSystems
- PASCAL/Z	Ithaca Intersystems
- UCSD PASCAL	SofTech

3.7 PL/I and Derivatives

- PL/I-80	Digital Research
- PLM-80	Intel Corp.
- PLMX	Systems Consultants, Inc.
- PLZ	Zilog, Inc.

3.8 Elimination of Candidates

The 22 languages noted above were studied further, some benchmarking was accomplished, and 13 of the 22 were eliminated. The reasons are provided below.

3.8.1 Slow

All of the interpretive, or compiler-interpretive combinations, were benchmarked for time and were found to be excessively slow. PASCAL/M, UCSD

PASCAL, SofTech FORTRAN-77 (part of the UCSD PASCAL system), and FORTH (and STOIC as being similar to FORTH) all required the range of 15-18 minutes to execute, compared to 1-4 minutes for the compiler languages. Such slow execution was deemed liable to have severe adverse effect on DCT use, both man-machine interaction and other factors, and the languages were eliminated from consideration. FORTH/STOIC were also undesirable due to the unusual difficulties encountered in trying to find a source of programming skill to code the benchmark program.

3.8.2 Limited Program Support Environment

Selection of H-P PASCAL 64000, or Intel PLM or FORTRAN-80, would effectively limit the PSE to that provided for by the respective vendors. The H-P PASCAL is very closely tied to the expensive H-P MDS, which offers greater capability than is required for the MDS portion of the PSE. The Intel MDS is competitively priced, but it is highly unlikely that an Intel MDS will ever support the NSC800. Since Intel is so well known, some benchmarking was done on the Intel languages. The preliminary results were not so good as to overcome the handicap of a limited PSE. In fact, significant problems were encountered in attempting to implement the Intel FORTRAN benchmark.

3.8.3 Problems

Various problems were encountered with ISO PASCAL, TSW RATFOR, and BDS "C." The vendor was not ready to release the ISO PASCAL, the TSW RATFOR was inferior to the Cromemco RATFOR and also did not work as advertised, and the BDS "C" was clearly targeted for the hobby market. They were dropped from consideration. There was also a problem with the Cromemco RATFOR in that Cromemco will not support the translator except on Cromemco systems. The RATFOR was evaluated nonetheless, since it was relevant to the evaluation of FORTRAN.

After additional study, it was determined that the Zilog and Cromemco FORTRANs were actually minor variants of Microsoft FORTRAN-80. They were dropped from consideration.

3.9 Final Candidates

After the analysis described above, there were 9 final candidates to be analyzed in greater detail, have benchmark programs executed, and have figures of merit calculated. The final candidates were:

- Interactive Systems C NOSC, INTERACTIVE
- Whitesmith C
- FORTRAN-80
- PASCAL/MT
- PASCAL/Z
- PLI-80
- PLMX SCI
- PLZ ANTHEM
- RATFOR MCTSSA

Of these, Interactive C was benchmarked at NOSC, San Diego, and at Interactive Systems, Santa Monica. PLMX was benchmarked at SCI, San Diego, and PLZ was benchmarked on a Zilog system at Anthem, San Diego (with a 2.5 MHz Z80). The remaining benchmarks were accomplished on an ALTOS computer system at MCTSSA.

SECTION 4. IMPORTANT LANGUAGE FEATURES

This section describes the features considered to be of importance in the analysis of the desirability of a language for the DCT. The primary features are operational features which directly affect the costs and benefits of the DCT. The technical features were evaluated in light of the importance of the operational features, and a figure of merit was developed based on scores and relative importance of the technical features.

4.1 Operational Features

Operational features of a language are those features or characteristics which contribute to, or detract from, three important considerations related to the success of the DCT project:

- Development Time and Cost
- Effectiveness
- Life Cycle Maintenance

The ultimate language selection criterion is that of using the language which will allow the development of a DCT that best supports the FMF. The operational features are those which are most directly related to this criterion, and which contribute to the development of the best DCT.

The paragraphs below address the three operational features and discuss their interrelationships.

4.1.1 Development Time and Cost

The more quickly the DCT can be developed, the better for the FMF. In addition, for any given budget, the less costly it is to develop or add any specific capability, the more capabilities can be included and hence a better DCT can be developed. This factor favors technical features which ease software development, make the project easier to staff, simplify or eliminate extra software tool development, and reduce efforts to force speed or space effeciency into the code.

4.1.2 Effectiveness

Although all the operational features in some way relate to effectiveness, this feature is specifically concerned with how well the fielded DCT works at any specific time. For example, ease of use of operator controls, speed of response, clarity of display information, number of functions incorporated into the system, error-free operation, and so on. This favors technical features which allow fast operation, allow necessary functions to fit in available memory, provide for good algorithms and man-machine interface, and tend to lead to error-free code.

4.1.3 Life Cycle Maintenance

Life cycle maintenance is concerned with two factors: corrective actions, the correcting of errors; and enhancement actions, the addition of capability. A large part of life cycle support is concerned with cost, but an even more important factor is the contribution of life cycle support to operational effectiveness, to the best use of the DCT. The easier the DCT software is to maintain, the faster will be the response to trouble reports and requests for modifications and enhancements. In addition, software that is extensible and easily modifiable can tend in the long run to be more compact and faster running. These factors are of particular importance for the DCT since preliminary indications are that the system will be quickly accepted and used, and many additional uses will be proposed. These factors favor technical features which lead to code that shows algorithms and data structures clearly, and is developed in accordance with software engineering principles such as high module cohesion and low coupling. In short, the technical features should allow for readable code and should support current MCTSSA programming standards.

4.1.4 Relationships

Each of the three operational features is closely related to development of the best DCT, and each of the three features is closely related to the other. For example, the more time that is taken in development, the more functions which may be added. If care is taken, the longer development may

also lead to more maintainable code. On the other hand, the same language features that lead to maintainable code lead to faster development and support operational effectiveness due to error-free characteristics. Contrary factors are also at work. Short development time and use of techniques to enhance readability could reduce the degree of effectiveness of an initial version of the DCT. On the other hand, emphasis on implementing as many functions as possible into the first versions could lead to very long development times and a system difficult to change. Such an emphasis could even be self-defeating in that the resulting DCT could be very error-prone.

In summary, the operational features and characteristics of a programming language are tightly bound to each other. The ways in which they are influenced by the technical features below, and the ways in which they contribute to the development of the best DCT, must be carefully considered by the analyst rating the technical features. In addition, consideration of these features is of great importance in scoring specific languages and in the final language selection.

4.2 Technical

Technical features or characteristics of a language are those which are important from the point of view of the system designer, programmer, and maintainer. They are inherent in the language and associated support system, ultimately support the operational use of the system to be developed, and collectively constitute the goodness or desirability of the language. The features are grouped as first order (most important), second, and third order (least important). The ordering of the features within first, second, and third order in the list below is unimportant. These features will be assigned weights reflecting their relative importance. Then each language will be given a score in each feature. Some of the features have "Score guidelines" to assist in scoring the language for the feature. A summary of the features is:

- First Order

  - Data Representation
  - Control Structures
  - Systems Programming
  - Program Support Environment (PSE)

- Second Order

  - Target CPU Transportability
  - Extent of Use
  - Learnability
  - Documentation
  - Time Efficiency
  - Space Efficiency
  - Assembly Language Linkage
  - Readability
  - NSC800 (Z80) Instruction Set Use

- Third Order

  - Multitasking
  - Reentrancy and Recursion
  - Compile-Time Efficiency
  - PSE Software Transportability
  - ROMable Object Code

4.2.1 First order. The following features are of most importance to selection of the HOL for DCT.

4.2.1.1 Data Representation

Data representation consists of declaration of data items by type, and their combination into more complex objects called data structures. A data structure is an object (representation) constructed in a regular way from component data items such as integers or characters. The most fundamental

data structure is the array. More complex structures are records queues, sets, trees, files, and combinations of the preceding. Greater flexibility and variety in possible data types and data structures are desirable attributes in order to naturally express relationships among variables. Specification of data types, compiler checks on incommensurate assignment of values (e.g., integer type variable := char type variable), and effective scope rules are also considered under data representation.

4.2.1.2 Control Structures

The inclusion of features which allow the code to be read from top to bottom is desirable. These features are the control structures usually associated with structured programming. Primary structures are the DO-WHILE and the IF-THEN-ELSE constructs, with CASE, REPEAT-UNTIL, and BREAK or EXIT (from the middle to the end of a control block) desirable additions. There must be a mechanism (BEGIN-END, brackets, DO-OD or DO-END, etc.) to allow multiple statements and multiple nesting of the control structures.

4.2.1.3 Systems Programming

Systems programming is the development and production of programs that have to do with translation, loading, supervision, maintenance, control, and running of computers and computer programs. Distinction is usually made between systems programming and applications programming. A HOL that supports systems programming should contain features which permit high-level input/output, so that device handlers may be totally written in HOL. The ability to invert, shift, rotate, and mask word, byte, and bit quantities is required, as well as the capability to handle interrupts and place values in absolute, specified memory addresses. A HOL that permits total control of a CPU at its machine code level is a systems programming language, as opposed to applications languages such as FORTRAN and COBOL. The ideal systems programming language would provide the ability to write a complex operating system without resorting to assembly language. There are three specific characteristics which distinguish a language which has systems programming capabilities as a real-time language. They are:

a. Bitwise Logical Operators. Operations should be allowed at the bit level on characters and integers to include: "and," inclusive and exclusive "or," left and right "shift" and "rotate," and unary one's complement or "not" function.

b. Memory Placement. It should be possible to place code (e.g., an interrupt handler) or data (e.g., to build, bit by bit, a data word for an I/O controller) at specific locations in memory.

c. Reentrancy. In order to handle interrupts properly, especially priority interrupt structures or in circumstances where two interrupt service routines mutually interrupt each other, it is necessary that common utility routines be reentrant.

4.2.1.4 Program Support Environment (PSE)

Although not specifically a language feature, a PSE suited for use with each language is an important consideration. The PSE should provide many capabilities in addition to the obvious one of source code compilation/assembly. An important component of a PSE, and for small projects the complete PSE, is a Microcomputer Development System (MDS). Both hardware and software components are involved; the hardware will be discussed first.

The nucleus around which the PSE hardware is constructed is a computer (either micro or mini or both), with its processor, input/output, and memory (RAM and ROM). In some cases the processing is not done in one processor, but several that perform individual, but related, functions.

An operator's work station is required to use the capabilities of the PSE. This includes a CRT for textual display with an associated standard keyboard. In addition, a special-purpose keypad is desirable for cursor center/up/down/right/left commands, as well as special commands to the operating system. Several multi-function (programmable) keys are also required for system-unique operator functions. The operator's work station may, or may not, have other devices such as computers, disk drives, Programmable Read Only Memory (PROM) programmers, or emulators, built into it. With the ever-increasing size of

microcomputer memory (therefore programs), multiple work stations have become increasingly necessary.

As in most cases where a CRT display is used, a printer should also be provided for hard copy of files, data, and listings.

Disk storage is required in a PSE. Two approaches are common. In the lower-cost MDS a floppy disk system allows for several hundred K bytes of storage. In more expensive MDS implementations, a hard disk device provides several million bytes of storage. The MDS should be readily adaptable to either storage approach, as may be required. A minicomputer-based PSE could provide hundreds of megabytes of storage, accessible by all programmers on a project.

A PROM programmer device should be a part of the MDS. This device includes sockets where PROMs can be inserted for blowing, comparing, copying, listing, or verifying.

An emulation capability is required with the MDS. It is used to check out code as though it were the actual target computer. While most MDS implementations provide emulators for several different microprocessors, only one (for the NSC800) is required for the DCT project. The emulator should be capable of operating at a speed of 4 MHz or greater.

Closely associated with the emulation capability is a logic analyzer. Its function is to remain transparent to the code being tested, while monitoring activities on busses and other key locations within the system. Instruction and input/output sequences can be traced with optional breakpointing.

An extensive set of software packages is required for use with the hardware just discussed. A sophisticated operating system is required to control the processor/peripherals, communicate with the operator, and provide required services. The software package most prominent in the view of the work station operator is the editor. A full-screen editor is required to take full advantage of the work station capabilities. The compiler for the language of the target processor is another necessary software package. It

must be available for on-line use by the operator. An assembler for the target processor is also required. While maximum use of HOL is desired, some assembly code will be essential. Another required software package is a link-loader for the object code generated by the compiler or assembler. This package completes the job of providing machine instructions executable in the target processor.

Thus far, software has been discussed that will allow programs to be created. That is, source files can be created; those files can be compiled or assembled into object files; and the object can be link-loaded into machine executable code. Many other software packages or tools are required. One example is a utility program that would transfer code from host to target computer. Another example is a utility program to convert BCD to binary code. An ideal PSE might include several dozen software tools; such a collection is sometimes known as a programmer's work bench. A minicomputer-based PSE provides the most extensive collection of software tools. The microprocessor-based MDS provides an important subset of those software tools, plus several others.

4.2.2 Second Order. The following features are not as important as the preceding.

4.2.2.1 Target CPU Transportability

why

A desirable characteristic is that the language, including compiler, support software, etc., should produce machine code that is loadable and executable on more than one CPU. Score guideline: 1 CPU = .50, each additional CPU, .1, up to 1.0 total.

4.2.2.2 Extent of Use

A widely used language is desirable from the point of view of training, acquisition, and retention of a programming staff. It is also important for increasing confidence in the compiler and other software tools, and providing a base for development of additional software tools. Score guideline: Remember there are two factors; the language itself (FORTRAN more widely used

than LISP) and the compiler, etc. (Microsoft FORTRAN used more widely than some other FORTRANs).

4.2.2.3 Learnability

Ease of learning is a desirable characteristic. Typical easy languages are BASIC, FORTRAN, and PASCAL. More difficult languages are PL/I and Ada. Score guideline: Rudimentary form of integer BASIC = 1.0. CMS-2 = (below .5). Consider that the factor relates to learnability of the language, not of programming. Assume the person learning already knows two other high order languages.

4.2.2.4 Documentation

Good, complete, error-free documentation is desirable. Consider the documentation of the standard language in general (say PASCAL), the documentation of the specific implementation of the language (say Zilog PASCAL), and the documentation of the associated support system. Consider ease of use, indexes, list of error messages, list of reserved words, concise syntax description, etc.

4.2.2.5 Time Efficiency

Time efficiency is measured by performance on the benchmark, i.e., how long the program takes to execute.

4.2.2.6 Space Efficiency

Same comment as for time efficiency. Criterion is the number of bytes of object code for the smallest (no I/O) executable version of the program.

4.2.2.7 Assembly Language Linkage

If a HOL does not support systems programming (see paragraph 4.2.1.3), then it is imperative that the HOL permit easy linkage (or inline code) to assembly language procedures. This is necessary to facilitate the machine

control necessary for the DCT application. The ease with which relocatable modules of compiled HOL are linked to assembled AL modules is important. Score guideline: If the HOL supports systems programming, then this capability is considered to be largely satisfied.

4.2.2.8 Readability

A language should be easily readable, even more than it should be easily writable. Readability depends upon many other things than the language, but factors such as data representation and control structures, mandatory statement labels, column restrictions, etc., can have a strong influence upon how easy it is to read the code. Commenting, free-form indentation characteristics, even the effectiveness of the editor for the associated MDS, can all affect the readability. Score guideline: Use the code generated for the benchmark as the primary basis for what is necessarily a subjective evaluation. Score guideline: Consider uncommented assembly code to be 0. A well commented, structured program in a block oriented language with careful data declarations is a 1.

4.2.2.9 NSC800 (Z80) Instruction Set Use

Since the NSC800, as well as the Z80, offers several significant architectural improvements over earlier processors, e.g., the 8080, the HOL should take advantage of the improvements. For example, the auto-increment type of instruction facilitates very efficient HOL looping. The HOL compiler that uses the instruction should rank higher in this factor than the HOL that does not. The same principle applies to all of the instruction improvements of the Z80 over the 8080. Score guideline: Use the generated code from the compilation of the benchmarks. If no Z80 unique instructions are generated, the language should receive a score of 0.

4.2.3 Third Order. The following features are of low importance for DCT HOL selection.

4.2.3.1 Multitasking

Multitasking, or concurrency, is the characteristic of a language which allows for either real or apparent simultaneity of processing. For example, PL/I allows the following code:

```
PROCA: PROCEDURE
       .
       .
       .
       CALL PROCB EVENT(EV1);
       .
       .
       .
       WAIT(EV2);/*SUSPEND UNTIL EVENT EV2, issued by PROCB*/
       .
       .
       .
       END;
```

PROCA and PROCB are presumably in two different processors. Ada uses "ENTRY" calls and "ACCEPT" statements to synchronize tasks. If a task issues an ENTRY call before the concurrent task is ready to ACCEPT, the issuing task suspends. Conversely, if a task reaches an ACCEPT prior to an ENTRY call, it is suspended. Often such features to support multitasking are provided by an operating system, if not inherent in the language. Score guideline: Effective procedures = 1.0, some capability = .5, could be easily implemented in operating system = .25, language offers difficulty in implementation = 0.

4.2.3.2 Reentrancy and Recursion

The property of reentrancy, exhibited by a program which contains no self-modifying features and uses dynamic (automatic, local, temporary, stack-oriented) data storage, is important to allow the sharing of processes by a number of user programs simultaneously. That is, one program may use the procedure without a prior use having run to completion. Recursion carries the

process further by allowing the procedure to use itself (directly or indirectly) as a subprocedure. Recursive procedures are often important for the efficient manipulation of dynamic data structures (such as linked lists and binary trees). Score guideline: Good facilities = 1.0, clumsy, but workable = .8, reentrancy without recursion = range of .5 to .8. No reentrancy = 0.

4.2.3.3 Compile-Time Efficiency

Compile-time efficiency is measured by the amount of time and memory taken by the compiler. This characteristic is considered to include time taken to link and load object programs. That is, it includes everything necessary to produce a (perhaps relocatable) load module or task image. For programs executed interpretively (the typical BASIC) or compiled-interpreted (the typical PASCAL or FORTH) special considerations are required. This feature will be measured by performance on the benchmark.

4.2.3.4 PSE Software Transportability

In some cases the support software used in an PSE may always remain in its host. More often, however, it becomes necessary to transport support software packages, e.g., a compiler, from one system to another within a computer facility. Therefore, it is highly desirable, where licensing permits, to be able to transport support software packages from one computer type to another. With this capability, especially valuable software tools used for the DCT project can also be utilized on the TAOC-85 project or other projects.

4.2.3.5 ROMable Object Code

The compiler should produce code that is suitable for use with read only memory. Specifically, the object code should not modify itself, and variable data and control information should be separate from instructions.

SECTION 5. NSC800 ARCHITECTURE

This section describes the architecture of the NSC800 in relation to the Z80 microprocessor.

5.1 NSC800 Architecture Description

The NSC800 is an 8-bit processor which has the following main functional areas: an ALU, register set, interrupt control, timing and control logic, and an internal 8-bit data bus.

5.1.1 Internal Architecture. The NSC800 has the same instruction set and timing as the Z80 and the same architecture with the exception of the following items:

a. The external bus structure of the NSC800 is a multiplexed address/data bus where the Z80 has separate address and data buses.

b. The NSC800 contains an 8-bit refresh counter as opposed to the 7-bit counter in the Z80. The refresh timing is identical, but the NSC800 will more easily support 64K dynamic RAM than will the Z80, with on-chip refresh logic.

c. The NSC800 contains on chip clock generation logic which only requires an external RC circuit or crystal.

d. The NSC800 supports three more interrupt signals than the Z80. This gives the NSC800 five levels of vectored, prioritized interrupts with no external logic required. Location X'BB is used for an interrupt mask, called the Interrupt Control Register which results in 255 unique peripheral I/O locations for the NSC800 vice 256 for the Z80.

5.1.2 Pin-out. The pin-outs of the NSC800 and the Z80 are the same except for the following:

a. Address/Data lines AD(0-7), provide multiplexed address and data on the NSC800 where the Z80 has separate lines for each.

b. Status lines (SO, S1) provide detailed information about the machine clock cycles on the NSC800 where there is only the M1 signal available on the Z80.

c. Input/Output/Memory (IO/M) is a signal which identifies if the current machine cycle is related to either an I/O or memory operation in the NSC800. There are two signals (MREQ and IORQ) in the Z80 which provide the same information.

d. The NSC800 does not have a HALT signal output, but uses the status signals (SO, S1) to provide the information.

e. Multi-Mode Interrupt (INTR) allows the NSC800 to operate in three modes of interrupt request. This signal in one of the modes is the same as the Z80's INT signal.

f. The following are additional pins which the NSC800 supports that the Z80 does not:

(1) Address Latch Enable (ALE) is used to demultiplex the address/data bus.

(2) Clock (CLK) is a clock signal generated by the CPU to be used as a system clock.

(3) Interrupt Acknowledge (INTA) is used to gate the interrupt response vector from a peripheral controller onto the address/data bus.

(4) Power Save (PS) is an input signal which suspends the interval CPU clock thereby stopping the CPU operation and halting accesses to other system components. The resulting power saving is an additional 50%.

(5) Reset Out (RESET OUT), is an output signal which indicates that the CPU is being reset and is normally used to reset peripheral devices.

(6) Maskable Interrupts (RSTA, RSTB, & RSTC) are input signals used for interrupts which can be controlled by software through the Interrupt Control Register.

5.2 Benchmark Impact

The NSC800 architecture will not have any effect on the benchmark of languages run on the same system. There is a possibility that software developed which makes extensive use of interrupts could run faster on the NSC800 than the Z80. The current benchmark does not, therefore the improved interrupt capabilities of the NSC800 are not used. The additional features of the NSC800 processor impact the hardware implementation of a computer by potentially allowing an implementation which requires less hardware logic than the Z80.

5.3 Language Impact

Due to the NSC800's extended interrupt capabilities, a language which requires interrupts will be easier to implement and execute faster. This will allow better control of peripheral devices, which will improve throughput.

5.4 Summary

The NSC800, as compared to the Z80, will have little impact on the benchmark or language selection. Any impact there is will be positive. The impact that the NSC800 will have is on the additional hardware logic required to implement a system. The use of the NSC800 will reduce the amount of external hardware required for a system when compared to a Z80 implementation. Some improvement can be expected in system performance with the use of the NSC800 over the Z80 due to the improved interrupt logic and simpler controller interfaces of the NSC800.

SECTION 6. BENCHMARK

This section discusses concepts related to the benchmark program, states the DCT programming problem upon which the benchmark and language evaluation were based, and presents the results of the benchmark process. The benchmark is designed to be a means of estimating programming language performance by measuring experiments on a computer.

6.1 General Approach

Much of a language evaluation can be accomplished by a static analysis, i.e. an evaluation of the characteristics and specifications of a language. However, such an approach does not yield as much insight into a language's nature as does actually writing code. Further, such an approach does not give any specific quantitative figures upon which to base a decision. There is little hope of evaluating the relative efficiency of various languages without executing a benchmark.

6.1.1 The Benchmark Process

Based on the DCT programming problem presented in paragraph 6.2, a computer program, the benchmark, was prepared in algorithmic or psuedo-code form. The benchmark is described in paragraph 6.3. The benchmark was then coded in each of the final candidate languages and executed under controlled conditions in accordance with the instructions of paragraph 6.4. Various items of information were gathered, the most important of which were execution time and program size, i.e. the number of bytes of memory required for the smallest (no CRT input or output) executable version.

6.1.2 The Benchmark Purpose

There were actually three reasons for coding the language in the benchmark program.

6.1.2.1 Comparable Figures

The most important reason is to have a program in each language that performs the same functions. Time and space efficiency statistics can then be gathered that reflect the capability of each language for the DCT application.

6.1.2.2 Language Example

The benchmark versions in each language served as examples of the languages. They were used as one source of language readability by the participants in the process of assigning scores to language for each technical feature.

6.1.2.3 PSE Use

In order to develop the code and execute the benchmark it was necessary to use the PSE to perform useful work. Such activity gives better insight into the PSE, and better ability to evaluate the PSE, than would simply reading or receiving a demonstration.

6.1.3 The Benchmark Programmers

Three different programmers coded the benchmark in a number of languages, 9 of which are provided in this report. There were, of course, differences in style and approach to implementation. An attempt was made to reduce these differences by adherence to the extensive comments and guidance in the benchmark program and by reading each others code. Each programmer attempted to code in the usual style of the language he was using. One language was programmed and executed by two programmers as a check on the impact of approach on the efficiency statistics. The variation was less than 10%, with one program being faster, but taking more memory, than the other.

6.1.4 The Benchmark Environment

Six of the languages were coded and executed on a microprocessor-based system at NCTSSA. Two others, PLMX and PLZ, used two similar but different

systems than the one at MCTSSA. Interactive C was developed on a minicomputer-based system, then loaded and executed in a Z80. With one exception, PLZ, the Z80s were Z80A with 4 MHZ clock and memory fast enough to ensure no wait states. During the final phase of the study emphasis was placed on Interactive C to ensure there were no hidden problems. Two different minicomputer systems and assemblers were used to generate code. The machine code was run on three different Z80 systems, with compatible results, including once with the program in ROM.

6.1.5 The Benchmark Statistics

A great deal of information was gathered in rough form. Some of the information was not readily available for all languages and hence not suitable for comparison. Other information was later judged to not be useful, while other information was useful in an informal sense during the integration analysis. The only information formally gathered and reported is that shown in Appendix C and consisting of:

- Execution Time.

- #Bytes Absolute Object Code (No I/O), being the measure of the size of the program.

- Compile Time.

These three figures are discussed below. The selection of the measure of program size was difficult and dependent upon some quite technical issues. The next three paragraphs assume a considerable degree of technical knowledge.

6.1.5.1 Execution Time

The value of execution time is straightforward. It is the time the program takes to execute.

6.1.5.2 Program Size

The best single value for program size or memory requirement is the "#Bytes Absolute Object Code (No I/O)," which is the smallest program created by the loader which will actually execute the benchmark processes. It will not, however, output to the CRT. This figure is partially dependent upon how effective the loader is in not picking up from the support library routines it does not require. Alternate values for memory requirement had been considered. If only the code size resulting from compilation of the benchmark source had been used, uncertain variability would have been introduced in terms of whether or not the compiler produced code in-line or called library routines. This figure was difficult to obtain in certain cases and would also be subject to doubt since the code could not actually execute. Another alternate was to use a figure that included the routines to perform I/O to the CRT. This, however, would unduly punish those languages, such as FORTRAN, that have very large I/O libraries which are linked when any I/O is performed. Such routines will not be required in the actual DCT. The advantages of the figure chosen for memory requirement are that it is provable since the program executes, and it is consistent across all languages.

6.1.5.3 Compile Time

The compile time is not a particularly important statistic, and is critically related to the quality of a compiler's error checking and diagnostics. Due to this factor and the fact that four different development environments were used, the compile time figures are not completely comparable. This was taken into consideration during the integration phase of the analysis.

6.2 Description of DCT Use

This paragraph provides a detailed description of the physical, communication, operational, and programming characteristics of the DCT, in order to provide to the reader a detailed background of the device, leading to an understanding of the benchmarks to be used in the study.

6.2.1 General

The DCT is a general purpose computer with specialized communication input and output. It allows the operator to compose messages, store them, recall them for review, and transmit them on call, and to receive messages, store them, and recall them for review. All procedures are aided by a menu select sequence which is software driven. The DCT display is a matrix of dot Light Emitting Diodes (LED). The input medium is a transparent pressure sensitive plastic film switch matrix superimposed over the display. Input and output is also accomplished over a set of communication interfaces described in detail below (para 6.2.3).

6.2.2 Physical Description

6.2.2.1 Case and Dimensions

The DCT case is high impact plastic. Its outer dimensions are 6-7/8" wide, 8-7/8" long and 1-7/8" thick, roughly the size and shape of a standard desk dictionary. The weight is four pounds, plus one-half pound for the canvas carrying case. Along the left side is a tubular handle which doubles as a holder for the primary power battery. The display is 3" by 4-1/2" and is composed of a 96 x 144 matrix of spot LEDs. A matrix of 54 switches embedded in a layered plastic film is superimposed over the display. Six special function switches are located below the display area. A short cable stub provides electrical connection to communication devices.

6.2.2.2 Internal Construction

The DCT circuitry is on three printed circuit boards (PCB) connected by a rugged flexible membrane which contains necessary interboard connections. The boards are embedded in shock absorbent material inside the case. Components on the three boards are grouped roughly in display, processor and memory, and communication groups.

6.2.2.3 Electronic Construction

The DCT is composed of electronic components in Complementary Metal-Oxide Semiconductor (CMOS) technology, chosen primarily for its very low power consumption. The microprocessor is the NSC800, manufactured by National Semiconductor Corporation, described in Section 5 above. The memory and communication components are standard chips manufactured into a set of hybrid leadless packages to enable close packing of components on the board.

6.2.3 Communication Characteristics

The DCT contains communication modems which interface directly with Marine Corps tactical communication equipment -- both radio and telephone -- and are compatible with communication security (ComSec) devices. Bit transmission rates from 75 bits per second to 16 K bits per second are selectable by the operator. Communication header and protocol fields are added by the software to each composed message, which may then be transmitted in a digital burst. The time of transmission is determined by the length of the message and by the transmission rate, but is typically in the range from 2 to 5 seconds. A suite of cables is supplied to connect the DCT to the various types of transmission equipment.

6.2.4 What the DCT Operator Does

This section describes how the operator initializes and checks the DCT, and how he composes, transmits, and reviews messages. The next section describes these actions from the detailed point of view of the software.

6.2.4.1 How the Operator Enters a Program Into the DCT.

Programs are entered into the DCT by a Program Entry Device (PED) manufactured by the DCT contractor. Programs are stored in the PED in non-volatile memory, and transferred to the DCT via the 188C Standard Interface under control of the bootloader program in the DCT, which is also contained in non-volatile memory. The operator cables the DCT to the PED and activates switches upon cue of the bootloader program. Program entry takes

about 90 seconds. It is also possible for a DCT program load to be made remotely over a radio or telephone, but this will be rarely done. It is very vulnerable to erroneous transmission and thus requires high error correction overhead.

6.2.4.2 How the Operator Checks the DCT

Present intent is that each DCT application program will include a self-test module. Upon selection of this action in the initial menu, the operator is led through a sequence of switch actions which exercise the display and switch matrix, and which allow a limited check of the communication interfaces.

6.2.4.3 How the Operator Initializes the DCT

When the power switch is turned on, the first menu presented will include a selection called 'SETCOM'. When SETCOM is selected, the operator initializes the communication parameters of: modulation bit rate, addressee information, and so on. This process is very brief and simple with the menu select features of the DCT.

6.2.4.4 How the Operator Composes a Message

The initial powerup menu will include selectors for each message type available. The message type desired is selected by pressing the selector symbol of that type. The next display will show an appropriate sub-menu on which further choices are made, until the fields of the selected message are to be filled in. At this point the display will show the fields and will also provide an input symbol array displayed with each symbol under a discrete switch location. A cursor will designate which field position is next to be entered. Each field is filled out in the order required by the message unless the operator chooses a different order by return to the menu. At any time the operator may also review his progress to that point in the composition by having the message displayed. He may continue or return to a previously composed field for change or correction.

6.2.4.5 How the Operator Transmits a Message

After a message is composed, it may be transmitted immediately, or it may be stored in a ready buffer in the DCT memory. When the operator wishes to transmit a stored message, he may call it from the buffer, review it for correctness if desired and then transmit by using the transmit switch at the display. Typically, there will be an automatic acknowledge (ACK) but this is a software feature and there may be circumstances where it is not used. If the operator does not receive an ACK, he may decide to transmit again. At present there is not an intent to have an automatic retransmit.

6.2.4.6 How the Operator Receives a Message

At any time the DCT is connected to a transmission device and is turned on, and after the correct communication initialization (SETCOM) information has been set, the device may receive an incoming message. This is done automatically, without operator intervention. An indicator light and an audio tone are activated to alert the operator to the fact of an incoming message. Typically, the DCT will be programmed to transmit a message acknowledgement automatically. The DCT will store the received message in an incoming message buffer, which is a portion of the main memory. When the operator wishes to review the message, he uses the appropriate switch to bring it to the display; if the message is longer than one display, the operator may scroll it through the display for convenient reading. The review process may also be used for recalling messages which have been composed and stored prior to transmission.

6.2.4.7 How the Operator Uses the DCT Peripheral Devices

Peripheral devices for the DCT include a Program Entry Device, a Map Generator Unit, and a printer. Each of the peripheral devices connects to the DCT via the communication cable, and each performs its function as if it were a remote station on a DCT communication link. The Program Entry Device (PED) contains a DCT program load in non-volatile memory. It interacts with the bootloader program in ROM in the DCT to transfer the selected program to the random-access semiconductor main memory of the DCT. The PED is enclosed in the same case as the DCT. The Map Generator Unit is a standard commercial

digitizer selected for its ease of militarizing should its use and fielding be approved following testing. The unit enables the operator to enter map features compatible with the limited graphics capabilities of the DCT and to transfer them into the DCT main memory where they may be modified by the DCT operator if desired and transmitted to another DCT or stored for later transmission. The printer gives a hard copy of messages received by the DCT.

6.2.5 What the Software Does

This section describes the software actions which accomplish the actions of the DCT described above. While it would be helpful to present the software in an organization directly paralleling the previous description of the operator view, it would be confusing and fragmentary to do so. The software is best described in the following organization: General, Hardware, Executive, Switch Matrix, Display, Communication Modems, 188C Standard Interface, and Miscellaneous.

6.2.5.1 General

The DCT software is input/output (I/O) intensive. It interacts in a complex way with the DCT hardware, and with the outside world by processing of digital bit streams in real time. There will be a constant evolution of the fielded software, and an exceptional volume of enhancement requests is expected from the Fleet Marine Force upon fielding of the DCT.

6.2.5.2 Hardware

A processor support hybrid module is the primary software point of reference to the hardware. Interrupts, I/O ports, timing, and control are terminated in this module for access and use by the processor. Each of the functional hardware subsystems -- switch, display, modem, 188C and miscellaneous -- is accessed through the processor support module.

6.2.5.3 Executive

Because the design of the NSC800 based DCT is as yet incomplete and the executive is not yet written, this discussion will of necessity be a series of educated guesses. Here -- actually throughout the software section -- it will be necessary to estimate how the programming will be done. In the existing 1802 DCT, much of the programming is driven by the architecture of the 1802 and by the macro-assembler employed by the designers. The executive will center on its requirement to coordinate access and execution of various tasks. The present design intent is to use four of the five NSC800 interrupts to handle directly the switch matrix, real-time clock, modem, and 188C interfaces, and to multiplex the remaining miscellaneous requirements by polling on the fifth interrupt line. Between key actions during message composition, for example, the processor will be able to accept an incoming message. Further, the executive will perform memory management, which involves control of the various buffers. The DCT is primarily an I/O processor -- 80% of the instructions for I/O is a good present estimate -- and the role of the executive in ordering the many possible transmit and receive combinations will constitute its main programming problem. There is a substantial need for concurrent programming techniques. For example, one of the specified requirements is to receive a message and dump it directly to the printer. There is considerable message decoding and reformatting required during this processing and strict synchronization is imperative.

6.2.5.4 Switch Matrix

The switch matrix contains 60 separate switches, of which six are fixed purpose -- display dim/bright, space and backspace, on/off, review, and message display. The remaining 54 are software addressable, and are typically keyed to touch points for menu select, to symbols to be input for character mode, or to points of input to graphics mode. Present design intent is to have the switch matrix decoded in a switch support hybrid module, the interface to which is via one of the NSC800 interrupt lines. Upon interrupt, the specific switch activated is determined over the switch module to processor interface, and the appropriate action for that switch is taken. This software has not yet been designed, but a CASE construct is obviously indicated.

6.2.5.5 Display

The display may be driven in character mode or in bit map mode. In character mode, the display is divided into 6 x 8 dot matrices. One side and the top or bottom are kept blank to provide spacing from the adjoining character. Each actual character is thus represented in a 5 x 7 dot matrix. The full display can represent 288 characters, which is 18 rows of 16 characters each. The basic data structure indicated here -- again prior to actual program design -- is the single dimensional array. However, a very memory-efficient method of driving the display in software is presently employed; a table of instructions is accessed and the instructions thus accessed are run to activate the LEDs required to represent the indicated character. There is a display support hybrid module in the DCT which is driven by the software to accomplish this function. The present intent is that the NSC800 DCT will use the same display support module. The other display mode is bit map mode. The entire display is mapped into 18 rows of 96 eight bit columns. Here there is substantial indication that an efficient two-dimensional array capability will be required.

6.2.5.6 Communication Modem

The principal programming issue in modem processing is efficient "shipping" of a bit stream to the modem. Since the processor ships bytes and the modem converts to a bit stream, there is an eight-to-one timing advantage for the processor over the modem for a given bit rate. This provides ample timing for the message control processing -- application and stripping of headers, parity checks, field delimiters and so on -- that must be done by the processor. This processing will require fluent manipulation of data at the bit level, and will be an important determinant of required programming language features. Since the direct memory access (DMA) features of the eventual design are yet to be determined -- and they may be suboptimal to the degree that costs force keeping the present 1802 design -- best guess judgments will have to suffice during this analysis.

6.2.5.7 188C Standard Interface

Cable connection to local DCT peripherals is supported by the 188C Standard Interface. The Program Entry Device, Map Generator Unit, printer, and Modular Universal Laser Equipment (MULE) interfaces all belong in this group. The principal programming concern for the 188C interface is the simultaneous receipt of a message from the modem and output over the interface to a peripheral. This may be done at dissimilar data rates, and the organization of memory and proper synchronization of apparently concurrent processes is a programming problem much eased by a language with concurrency features. In addition, I/O and field conversion problems similar to those for the modem exist.

6.2.5.8 Miscellaneous

The DCT also contains an audio alarm, and a real-time clock which must be dealt with in software, but there are no unique problems presented by this need.

6.2.6 Summary

From the preceding discussion, the following essential characteristics of the DCT programming problem emerge:

- Exceptionally heavy I/O.
- Important concurrency requirements.
- Heavy state transition control requirements (CASE construct).
- Bit level processing of communication header and protocol data -- often at high rate and desirably in real time; that is, without extensive buffering.
- Language compatibility with the DCT as a computer system with specially designed hardware interface.

- Extensive table processing -- efficient look-up but not additions or deletions.

From these programming characteristics, the following features of a programming language appear to be desirable:

- User definable and efficient I/O, as in "C".

- Concurrency features, as in Ada or Concurrent Pascal.

- CASE construct, preferably with an 'OTHERWISE' clause.

- Sub-byte constructs, as in PL/M; access to registers.

- A "tunable" compiler, with the source code available to the government for purposes of tailoring.

- Good array and string handling.

6.3 Benchmark Program

The benchmark program is a "synthetic" benchmark. It is not actually DCT code nor does it attempt to exactly replicate the mix of instructions in current DCT programs. Rather, it performs functions such as procedure calls, psuedo message processing, output to a port, and a complex logical branch, that are typical of DCT applications. The relative importance of the various types of operations is dependent upon the selection of loop control variables. Selection of these variables was based on a consensus of opinion of those at MCTSSA familiar with the DCT programming problem. The benchmark itself is presented in Appendix B in algorithmic form, followed by its implementation in each of the 9 languages.

6.4 Benchmark Instructions

This paragraph presents the instructions given for the gathering of statistics during execution of the benchmark programs. The tense and person

of some of the paragraphs is indicative of the fact that these were the actual instructions. Of all the data scheduled to be gathered, some was unavailable, some was incompatible from language to language, some was of value only in support of the subjective analysis, and some was later determined not to be required. The only officially gathered data are the results presented in Appendix C.

6.4.1 Benchmark Versions

There are a number of reasons for implementing the DCT benchmark program. Some of these reasons are to obtain timings for compilation and execution of the benchmark, to obtain sizes of compiled object code, and to evaluate the compiler's capacity for handling compile-time errors. In addition, the code produced for each language was used during phase II of the Delphi Study. The language analysis was largely based upon the samples of each language provided by the benchmark. In order to obtain this information, three versions of the benchmark are required.

6.4.1.1 Code Only Version

This version of the benchmark contains the benchmark logic presented in algorithmic form, in a programming design language similar in appearance to PASCAL. The uncommented listing is for your convenience and is available on a CP/M diskette, ready for adaption to several of the languages.

6.4.1.2 Heavily Commented Version

This version is the same as the code only version, except that it contains extensive comments to guide programmers in language-specific implementations.

6.4.1.3 Error-Seeded Version

This version of the benchmark is the same as the heavily commented version except that several errors have been introduced to cause compile-time errors for the purpose of evaluating compiler error-handling capabilities.

There is a listing provided. Implement this version after you have correctly implemented the normal version. Use a text editor to put the errors into a copy of your source file(s) for the code only version.

6.4.1.5 I/O Free Version

This version of the benchmark is the same as the code only version except that the two writeline lines of code have been removed. This is done to assess run-time routine sizes. Create an implementation of this version from a copy of your implementation of the code only version. Simply edit out the statements which implement "writeline."

6.4.2 Instructions for Coding

Please read these instructions and the comments in the program listings before you start coding:

- Assembly language should be used only to implement functions which cannot be implemented in HOL. Assembly language functions/procedures should be coded as general purpose functions/procedures and should not be coded in-line. Use of assembly language solely to achieve efficiencies over HOL is prohibited.

- In those languages, such as FORTRAN, which do not support the CASE construct, the use of repeated IF statements is allowed.

- In each language, the benchmark should be implemented in an efficient manner, taking advantage of language features if possible, as long as other rules are followed.

- The coding should reflect appropriate style for the implementation language. An important use of the benchmark is to allow comparisons of language features.

- If possible, link only those run-time linkable routines that are necessary for successful benchmark execution.

6.4.3 Instructions for Compiling the Benchmark

Before compiling the benchmark, sections A through J should be completed on the benchmark report. Next, compile the benchmark (code only version) without obtaining listings or cross references, etc. Next, compile the same version obtaining maximum documentation support. Denote all compiler switches set. Make entries K, L, and M. Compile the I/O free version, then fill in sections N through W.

6.4.4 Instructions for Executing the Benchmark

Execute the code only version of the benchmark. Begin timing when the first letter of 'BEGIN EXECUTION' appears. Stop timing when the last letter of 'END EXECUTION' appears. Log this time at location X.

6.4.5 Instructions for Compiling the Benchmark with Errors

Insert the errors stated in the error-seeded version of the benchmark. Compile the benchmark, noting time to compile at location Y. Answer the question of location Z and discuss the error messages at AA.

6.4.6 USMC DCT Benchmark Report

A: Language: _____ B: Vendor: _____

C: Analyst: _____ D: Date: _____

E: MDS: _____ _____ F: Location: _____

MDS Notes:

G: CPU Clock Rate _____ H: Memory Speed _____

I: Operating System _____ J: Comments _____

If cross-compiler used, describe host on reverse.

K: Compile time (fastest compilation): _____

L: Compile time (maximum documentation): _____

M: Other compile times, if pertinent: _____

N: Link/Load time (under CP/M use submit file method): _____

O: Total compile and Link/Load Time: _____

P: Compiler size (bytes): _____

Q: Link/Loader size (bytes): _____

R: Object code generated by the compiler from compiling the I/O free version (number of bytes of instructions): Relocatable _____ Absolute _____

S: Object code generated by the compiler from compiling the code only version (number of bytes of instructions): *Relocatable* _____ Absolute _____

T: Was a relocatable step included? _____

U: Were operating system or system support routines linked? _____ Explain. _____

V: Absolute object code including run-time routines to be loaded. Does not include array storage. Obtain from compiling I/O free version: _____

W: Absolute object code including run-time routines to be loaded. Does not include array storage. Obtain from compiling code only version: _____

X: Execution time of the code only version: _____ (optimized for run-time efficiency)

COMPILER ERROR DETECTION

Y: Compile time (of the error-seeded version): _____

Z: Were all errors discovered? _____

AA: Describe all error messages: _____

6.5 Results

The results of benchmark testing are provided in Appendix C and summarized in Figure 6-1 on the next page. Note that low numbers are preferred for both measures presented. In execution time, Interactive C is best, with PLMX and Whitesmith C being close behind. FORTRAN and RATFOR were the slowest. For program size Interactive C and PLMX were best, while the largest programs were produced by PLI-80, RATFOR, and FORTRAN.

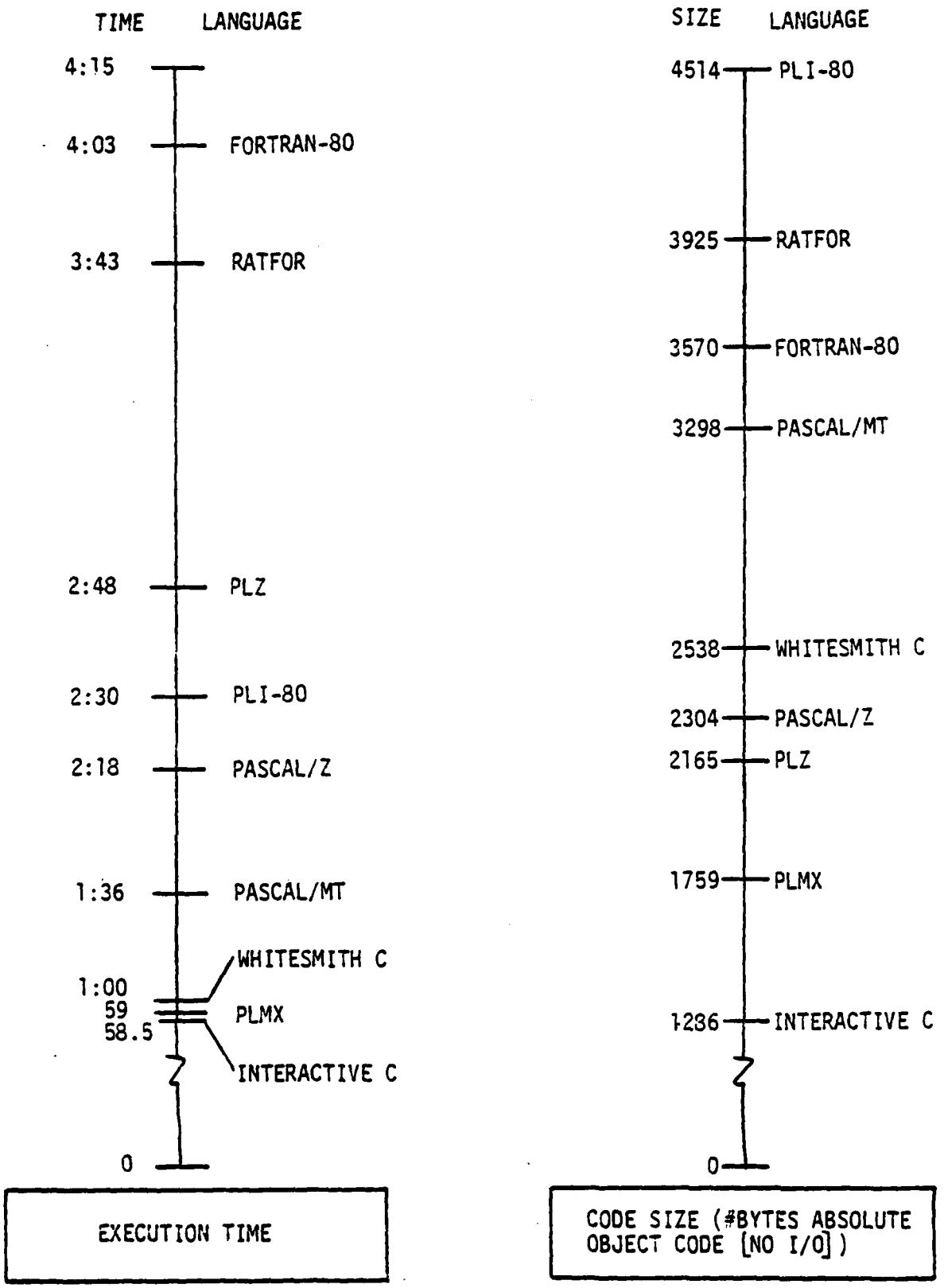

Figure 6-1. Benchmark Results

SECTION 7. FIGURE OF MERIT METHODOLOGY

This section presents the methods and procedures used to determine the figure of merit for each language. It includes the assignment of weights to the 18 technical features of HOLs.

7.1 General Approach

In order to assess the relative desirability of the languages and associated support systems, it is appropriate to obtain a single number, a "figure of merit," for each language. In order to do so, 18 specific technical language features were specified and defined. Each of the features was assigned a weight reflecting its value or importance relative to the other features. Then, each of the candidate languages was assigned a score between 0 and 1.0 which reflected the degree to which the language was good, effective, or appropriate in regard to each technical feature. The sum of the product of the scores and weights was computed for each language in order to determine a figure of merit. The weights and scores were assigned by technical computer analysts at MCTSSA and NOSC familiar with languages, microprocessor characteristics, and DCT real-time applications. The weight for each feature and the scores for each language were derived as the average of the weights and scores assigned.

7.2 Weighting and Scoring Methods

Weights and scores were assigned by a group of analysts according to the instructions described below and using the methods described in paragraph 7.3. The tense and person of paragraphs 7.2 to 7.5 reflects the fact that they were used as instructions to the evaluators.

7.2.1 Weights for Language Features

Your task is to assign weights to the technical features listed and described in Section 4. The weights are to represent the relative importance of each feature as it specifically relates to the DCT programming problem described in paragraph 6.2. Your assignments will be confidential, and you

should not discuss your assignments with other people participating in the study. The assignment of weights will provide a ratio scale of value of the features. Any weight assignments that you feel are likely to be considered unusual, extreme, or otherwise surprising, should be justified with written comments. There will be a Delphi-like iteration as described in paragraph 7.3.

7.2.1.1 Initial Weight Assignments

You have 1000 points to distribute among the features. You may think of them as dollars which you can spend to buy a certain amount of each feature of a language to support the DCT software development. Carefully assign what you feel is an appropriate weight to each of the features, completely using up the 1000 points.

7.2.1.2 Reassignments for Consistency

It is important that all of the assignments or purchases of value be consistent. For each group of 3 features, say A, B, and C, perform the following checks. If you would rather have feature A than both B and C, make certain that the weight assigned to A is greater than the sum of the weights assigned to B and C. Of course if B and C are preferred to A, their weights must be more than the weight of A. This is not an easy task since the consistency must extend to groups of 4, 5, 6, etc. Do the best you can to be consistent among all combinations of features, making reassignments of weights until you are satisfied.

7.2.1.3 Initial Guidelines

Section 4 provides three groups of features, depending upon their relative importance. In your rank order of features by importance, the first order features should be at the top and the third order features should be at the end of the list. If you feel this initial grouping is incorrect, rank the features the way you believe is right, and provide written comments to explain your feelings.

7.2.1.4 Methodology Notes

While assigning the weights, you may notice or have the feeling that certain features are not mutually exclusive, that one feature depends upon another, or that values of certain features are not additive. Another methodological issue, but one which would be difficult (and in some cases impossible) to resolve, is that not all evaluators will evaluate all languages. This is related to the general issue that the weights and scores are dependent upon the analysts participating in the Delphi. Finally, the actual relative importance of the features depends not only on the weights assigned, but also upon the relative variability of the scores. Factors such as these are why the method of determining a figure of merit is only approximate, and that the final value will be only one factor in the determination of the HOL for the DCT. If you feel that any of these considerations bias the results of the analysis, you should submit written comments with your weighted values for each feature.

7.2.2 Scoring of Specific Languages

Your task is to assign scores to each of the final candidate languages listed in Section 3. Each language is to be given a score for each of the 18 technical characteristics listed in Section 4. Complete documentation is available at MCTSSA. Do not return scores for any language with which you do not feel knowledgeable. Each language is to be scored on a scale from 0 to 1.0, with high values meaning it is good, desirable, effective, etc. Take care to use the entire range of 0 to 1.0. Use the following general guidelines. Excellent = .9; Good = .5; Poor = .1; Totally Unsuitable = 0. A score of "1" means it is the best it can be. The scores are to be assigned with the specific DCT programming problem described in paragraph 6.2 as a goal. Paragraph 4.2 contains some guidelines for scoring for your optional use. Any score assignments that you feel are likely to be considered unusual, extreme, or otherwise surprising, should be justified with written comments. There will be a Delphi-like iteration as described below.

7.3 Delphi Approach

After submission of weights or scores by all participants, the values will be averaged, histograms to show the deviation in weight values for each feature will be prepared, written comments will be summarized and integrated, and all the information will be returned to each participant. In a second iteration of the analysis, you will then be asked to reevaluate your initial assignments in the light of what other analysts have thought about the same features. Your assignments and comments will be kept confidential, not revealed to other analysts participating in the study. You are asked to not discuss the assignments with others, but do take the group opinion into account when doing a second assignment of values. If you have any comments on the description of language technical features, upon the score guidelines as part of the feature description, or upon these instructions, submit them with your values for the first iteration of the analysis. Such comments will be taken into account for the second iteration. The process described is known generally as the "Delphi" process, pioneered and popularized by Rand Corporation in the late 1960's. The essential feature is that of iteration and individual changing of perceptions based upon what others feel about the same situation. The process tends to lead to a group consensus.

7.4 Figure of Merit Determination

The figure of merit (FOM) for each language will be derived in the following way. Determine the weights and scores as the average of the values assigned by study participants during the Delphi analysis. For each language feature, multiply the weight given the feature times the score given the language in that feature. Sum up all the scores. The result, with a perfect score being 1000 points, is the figure of merit. Algebraically,

$$FOM = \sum_{i=1}^{18} \text{Score}_i \times \text{weight}_i .$$

7.5 Instructions for Delphi. There will be two phases in the Delphi analysis. Phase I deals with assignment of weights to language features.

Phase II deals with assignment of scores to each language, for each feature. Instructions for each phase are outlined below.

7.5.1 Phase I: Weights

You will receive four handouts:

- Language Feature Description (Section 4)
- Description of DCT Use (paragraph 6.2)
- Figure of Merit Methodology (Section 7)
- Weight Worksheet

Following topics were discussed at meeting of first iteration:

- Background of Study
- Nature of Delphi
- Description of Handouts
- Discussion of Operational and Technical Features
- Specific Instructions

7.5.2 Phase II: Scores

- Assign scores as indicated in paragraph 7.2.2

7.6 Summary

The methodology described in this section was followed to derive weights for the 18 technical language features and to derive figures of merit for 9 final candidate languages. The weights for the features are shown in Figure

7-1, and in Appendix D, and the language feature analysis is presented in Section 8.

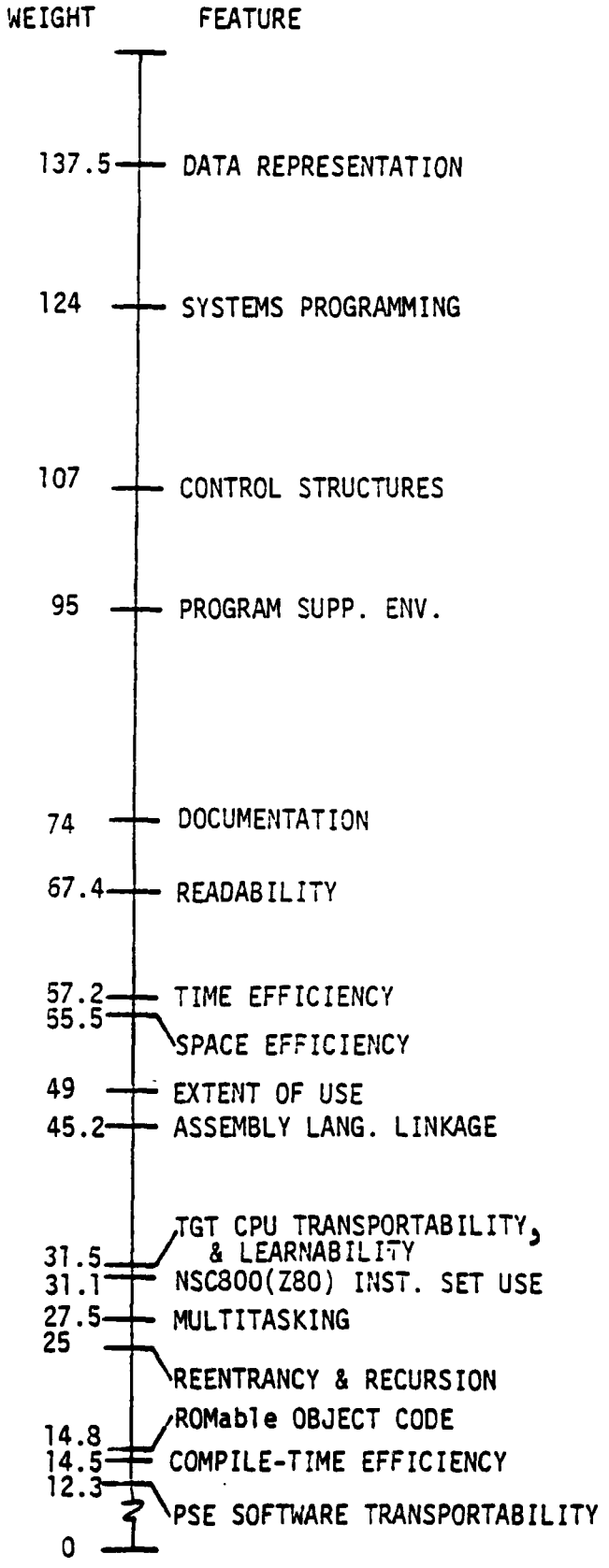

Figure 7-1. Ranking of Features

SECTION 8. LANGUAGE FEATURE ANALYSIS

This section summarizes the language feature analysis. Some factors common to all the languages are discussed, then each of the nine final candidates is reviewed. Interactive C, as the language selected for DCT programming, is discussed at greater length than the other languages and is occasionally mentioned during discussion of other languages. For each language the only features reviewed are those for which something is unusual or important. The scores received by each language for each feature, and the overall FOMs, are presented in Appendix E. Paragraph 8.10 summarizes the results. The discussion focuses on the scores received during the FOM derivation, but also discusses some unquantifiable topics and some items of information that became available after the Delphi analysis was complete. For purposes of comparison of languages, PASCAL as defined in the Jensen-Wirth Report is used as a baseline. For this section only, it is assumed the reader has a working knowledge of PASCAL. This section also assumes a greater level of computer science and microprocessor expertise than does the remainder of the report.

8.1 Common Factors

This paragraph addresses features for which a general comment satisfies all or most languages. In some instances, the scores assigned by the Delphi were inconsistent with known or agreed upon facts, or incompatible with the subjective opinions of the principal investigators. Such cases are pointed out below. None of these inconsistencies or incompatibilities affected the scores to a great degree. They were accounted for during the integration phase of the analysis.

8.1.1 Data Representation

Data representation is discussed for separate languages, but some general comments are appropriate. All the languages except FORTRAN and RATFOR have some facility, such as pointers, for the construction of dynamic data structures. All languages implement functions or procedures or subroutines which allow the passing of parameters. FORTRAN allows only the feature of

pass by reference. Some other languages only allow pass by value, but a pointer can be passed to effectively implement a pass by reference. A useful capability is to be able to interpret the same part of memory in different ways, say as either two integers or four characters. "C," FORTRAN/RATFOR, and PASCAL allow this capability through the use of, respectively, "UNION," "EQUIVALENCE," and "VARIANT RECORDS." PLI-80, PLMX, and PLZ do not explicitly allow this capability. All of the languages handled floating point numbers except Interactive C, PLMX, and PLZ. Several of the languages, Whitesmith C, PLMX, and PLZ, either had problems with or were not designed to handle negative numbers.

8.1.2 Documentation

There was little perceived difference in documentation, with "C" having a slight edge and PLMX scoring slightly lower.

8.1.3 Time and Space Efficiency

Time (execution time) and space efficiency features were evaluated objectively, based upon benchmark results. Space efficiency was measured by number of bytes of absolute object code, including data storage but not including I/O routines. Both of the "C" languages did better (by several seconds and perhaps 100-150 bytes of code) than they might under other circumstances. The reason is that the arithmetic in the benchmark involves the manipulation of constant values. The "C" compilers were complete enough to recognize this and do all the calculation at compile time. This speaks well for the compiler, but affects the statistics. It is worthy of note that there was a great deal of variability in the efficiency scores. Section 6 has presented the benchmark results.

8.1.4 Target CPU Transportability

The languages generally fell into three categories of code generation: Z80 only, 8080 family, and 8080 family plus others. Even those languages which were Z80 only, Interactive C, PASCAL/Z, and PLZ, received some points. This was in accordance with the guidelines and perhaps is an indication of a

perception that the transportability would not be difficult. Note that a compiler which scores low in this feature by generating code specific to the Z80, should score higher on the feature NSC800 (Z80) instruction set use.

8.1.5 Learnability

There were few differences in the learnability scores, with the PASCALs having a slight edge. The fact that PASCAL doesn't score even higher, despite having been designed as a teaching language, reflects the fact that PASCAL/M and PASCAL/Z are implementation languages. They have extra, useful, features which go beyond the Jensen-Wirth definition.

8.1.6 NSC800 (Z80) Instruction Set Use

The compiler either did, or did not, use Z80 special instructions. Some of the score variability could result from looking at the actual code generated by the compilation, but some appears to be either wrong judgement or a perception that such code could be generated by small compiler modifications. The compilers which can generate Z80 code are Interactive C, both PASCALs, PLMX, and PLZ.

8.1.7 Multitasking

None of the languages specifically supported multitasking as defined in Section 4.

8.1.8 Reentrancy and Recursion

Reentrancy and recursion is largely a yes or no issue, although there are matters of technique and a partial capability for PLI-80. The languages with full capability are both "C"s and PASCALs, and PLZ.

8.1.9 ROMable Object Code

All languages produced code which could be placed in ROM, but evaluators were given the guidance that it was somewhat more difficult in Interactive C,

FORTRAN-80, RATFOR and PLI-80. Other score variations reflect the relative ease or difficulty of methods of using the loader. A late result, after the Delphi, was that Interactive C is easily ROMable. In fact, Interactive C was benchmarked with compatible results on three different processors, including one series of executions with the code and fixed data (a jump table) in ROM and variable data and the stack in RAM.

8.1.10 Compile-Time Efficiency

These figures were taken from benchmark results. There are some incompatibilities which were taken into account during the integration phase. The PLMX and PLZ benchmarks (and compilers) were run on a different MDS than other languages, and the Interactive C was compiled on a minicomputer. The PASCAL/MT compiled quickly, but did not provide intermediate assembly language code and had poor error diagnostics. The PLMX and PLZ would have compiled faster on the system at MCTSSA. However, there does seem to be a general inverse relation between compile time and execution speed (e.g., PLMX and Whitesmith C), perhaps reflecting the fact that the compiler is working a great deal to produce efficient code. In one sense the Interactive C provides the best of both worlds: considerable code optimization, but yet fast compilation since the system runs on a powerful minicomputer and uses a PDP-11/70 to Z80 cross-compiler. The 45 seconds for compilation is a typical figure. The time varies, both higher and lower, depending upon other processing loads on the multi-user system.

8.1.11 PSE Software Transportability

This is a minor issue with little variability. Interactive C and Zilog are properly the lowest scorers.

8.2 Interactive Systems C

8.2.1 Data Representation

"C" has an appropriate, modern set of structures for data representation. It is similar to PASCAL with the major exception of no type definition or type

checking. Most analysts consider the lack of type checking to be a deficiency, although it occasionally makes programming easier when it is actually necessary to compare or assign incompatible types. That is, strong type checking is not an unmixed advantage. There are no floating point numbers in the Z80 implementation. There is a "typdef" in "C", but it is for programmer convenience and program readability. It does not create a new type. "C" uses the word "function" rather than procedure, and a function may or may not return a single value. "C" has dynamic storage for local variables and can be recursive/reentrant. All parameters are passed by value, but arrays and structures appear to the programmer to be passed by reference. A pointer to the array or structure is actually placed on the stack. "C" is not a block-structured language in the sense of nesting procedures within procedures. Within a separately compilable module, all functions are global. However, "C" has most of the advantages of a completely block structured language since most variables are likely to be dynamic (automatic), static variables can be declared "private" to a function, and modules should be small. "C" allows a physical top-down construction of a module, with the main function at the top and successively subordinate functions following in a logical order. "C" allows local dynamic variables to be declared as "register" variables as a hint to the compiler to attempt to use registers for increased speed.

8.2.2 Systems Programming

"C" was designed as a systems programming language. The UNIX operating system and the "C" compiler are written in "C." Bit-wise "and," "or," and "not" are part of the language, as is a shift operator, ">>." It is a relatively small language, but yet with a great many operators, e.g., ++index is index:= index +1 with the increment occuring before subsequent operations within the same statement. The language was designed with the PDP-11 architecture in mind, but also to generally result in efficient code under any circumstances. The benchmark results show that the compiler produces efficient code for the Z80.

8.2.3 Control Structures

"C" has all necessary, and PASCAL-like, control structures including a "break" for loop exit and a "default" in the case statement. "C" uses brackets to denote a block-like span of control. The language designers feel that brackets contribute to readability by being less obtrusive than "BEGIN-END" pairs. However, for those who desire to do so, it is possible to "#define" the brackets as "BEGIN" and "END." The resulting code can be very PASCAL-like in appearance.

8.2.4 Program Support Environment

Interactive Systems C scored lowest of all languages in this factor, exactly contrary to the feeling of the principal analysts on the study. It is likely that others' unfamiliarity with the cross-compiler and minicomputer-hosted PSE led to the assignment of low scores. In fact, one of the very nice features of Interactive C is the fact that it is integrated into a sophisticated PSE. It would be possible for example, to write a module of code, compile with the PDP-11 as the target machine, execute and debug on the PDP-11, then compile targeting the Z80 *and* down-line load for final testing. This process was followed during benchmark development. Furthermore, "C" operates more effectively than would other languages (since it is designed as part of the system) with the Interactive Systems PWB/UNIX operating system, and Source Code Control System (SCCS). A software tool such as a SCCS is mandatory for development of large systems, with many programmers and many versions and releases. There is also a program "Make" facility, unfortunately independent of the SCCS, which allows the automated construction of new versions or releases of programs. Interactive C, since it is implemented as a cross-compiler, works well with the Make facility. In short, the PSE is a very strong point in favor of Interactive C.

8.2.5 Readability

"C" has a potential to be somewhat less readable than PASCAL. It has many operators, allows very concise and information-packed statements, and allows freedom in type conversions. Some care must be taken in adherence to

standards however, since "C" can be made concise to the point of being cryptic. Also, the usual "C" style of lower-case letters for variables and reserved words, with CAPS only for constants, looks strange to those used to other more common styles. Of course, one can adopt other conventions to make "C" look like PASCAL. "C" does have some advantages also. Once one is used to the operators, statements such as:

```
if ((index++) != LIMIT)
```

are more concise and meaningful than the PASCAL equivalents. There is considerable flexibility in program layout, excellent capability for top-down coding, and effective use of variable declarations with initial values and of "#define" for constant definition. The evolution of developing a "C" program is physically "top-down" and thus somewhat easier to read and write than PASCAL from this standpoint. Top-level modules are physically first in the source code file.

8.2.6 Extent of Use

"C" is not as widely used as PASCAL, but is a popular systems language. If a programmer does know PASCAL, he can be very effective in "C" quite quickly. The Interactive C and most of the PSE is a supported version of the Bell Lab's system with the cross-compiler originally developed at MOSTEK. Interactive Systems has sold over 60 of their complete systems, although only 5 also have the cross-compiler. One of the 5 however, is NOSC, San Diego. They are using and supporting their own slightly modified version of the cross-compiler. Interactive C received a low score in this feature, properly so, but the user base is more than adequate to ensure compiler use and programmer availability.

8.2.7 Assembly Language Linkage

Interactive C is clearly superior in this feature. It is the only language which allows either the use of direct, in-line assembly language, or linking to an externally defined assembly language routine. It passes parameters on the stack and gets a return function value in the Z80 H-L

register pair. A negative factor is that the standard assembler expects the Bell Lab's mnemonics, which are not well designed for human use. It is not a macro-assembler. The documentation on assembly language linkage is clear, contrary to the experience suffered with a number of other languages, and implementation was easy.

8.2.8 ROMable Object Code

After the Delphi was completed, the benchmark was executed with the constant program (710 bytes) and data (a 20 byte jump table) in ROM, and the variable data (556 bytes) in RAM. No special procedures were required to separate program and data. If this information had been known earlier, the language would have scored higher on this feature.

8.2.9 Summary

"C" is an easy language to learn and use. It allows the physical layout of the program to be "top-down." The Interactive Systems PSE makes implementation efficient. It is an excellent systems programming language. It is efficient and is suitable for large programming projects.

8.3 Whitesmith C

Most of the comments made for Interactive C are applicable here. The major differences are that Whitesmith C does not have such an extensive PSE, there are more users of the specific compiler, and it does not allow direct, in-line assembly language code. In some testing separate from the benchmark, wrong answers were produced in expressions involving a combination of large negative and positive numbers. Whitesmith also supports a PDP-11 to 8080 cross-compiler which generates the same object code as the Z80-hosted compiler used for the benchmark. The support environment is not as complete a PSE as the Interactive Systems PWB/UNIX.

8.4 Microsoft FORTRAN-80

8.4.1 General

FORTRAN was the first popular HOL, defined and developed in the mid-50's. It is a primitive language, but with the ability to express equations in the format of a formula. It was defined before modern concepts of software engineering were formed and before the difficulties of many programmers working on a large project were well understood. Nichols, in reference (f), notes that "the importance of string handling was not widely appreciated during the period when programming languages were first being developed. Thus languages such as FORTRAN and ALGOL have virtually no string processing capabilities." The language has served well and still has important applications for small numerical programs, but is unsuitable for large programs with little emphasis on numerical computations. The paragraphs below discuss some of its weaknessses.

8.4.2 Data Representation

FORTRAN does not have records/structures, variable arrays, or pointers. It does not handle characters well and has limited bit manipulation facilities. It has no dynamic data structures at all. It does not allow a programmer defined linked list, tree, or similar structure, nor does it provide for automatic allocation of variables in a routine. It passes parameters only by reference and has a cumbersome method, "COMMON," for controlling scope of variables. All subroutines have global names, making hierarchical program design difficult to enforce. It allows type incompatible assignments and does not allow type definitions. It allows use of variables without declaration. Reference (i), beside claiming that FORTRAN is unportable, also comments in regard to storage and handling of characters, "a task for which FORTRAN was not designed and about which the standard is deliberately vague." Reference (c) notes a number of problems, especially that "arrays must be fixed in size, regardless of the actual needs of the application." Languages with pointer variables can overcome this limitation. The limited data representation capabilities force a restrictive and artificial style for complex programming problems.

8.4.3 Systems Programming

FORTRAN's low score in this feature reflects that it is simply not designed to manipulate devices and blocks of data. It is designed to perform numerical calculations. Its primary lack is in the area addressed above on data representation. Microsoft FORTRAN-80 does provide additional bit manipulation capabilities than standard FORTRAN, but then the code no longer follows the ANSI-66 standard.

8.4.4 Control Constructs

FORTRAN's control constructs are the satisfactory "DO" statement for iterative looping, and the unsatisfactory "GOTO" a label for all other circumstances. The dangers of the "GOTO" and use of labels are well known. FORTRAN does not provide the constructs for "structured" programming, the IF-THEN-ELSE and DO-WHILE, and does not allow code to be written in accordance with Mil-Std 1679. In discussing the lack of effective control structures, reference (g) states: "as a result, the form of a FORTRAN program cannot ordinarily be made to reflect very clearly the control structure of the underlying algorithm..." The consequence of this is that for complex problems FORTRAN programs can be difficult to read and understand, error-prone, costly to produce, and hard to modify or expand.

8.4.5 Readability

Due to the problems of data and control structure, as well as clumsy formatting rules, comment and line continuation conventions, and numbered labels, FORTRAN programs are difficult to read. A relatively recent insight is that a computer language must be designed (as was Ada) for ease of reading rather than writing. A program is read much more often than it is written.

8.4.6 Extent of Use

Both FORTRAN in general and Microsoft FORTRAN-80 are widely used.

8.5 RATFOR

RATFOR is FORTRAN with control structures and an improved commenting convention. The addition of "IF-THEN-ELSE" and other flow of control statements makes RATFOR much superior to FORTRAN for readability. The disadvantage is that the code must first be pre-processed before being compiled. This both takes extra time and creates difficulty during program test and debug, since errors are related to the FORTRAN rather than directly to the RATFOR source file. Otherwise, RATFOR shares the deficiencies of FORTRAN.

8.6 PASCAL

The two PASCALs can be discussed together with few exceptions.

8.6.1 Data Representation

As expected, the PASCALs received the highest scores in data representation.

8.6.2 Systems Programming

PASCAL/Z scored lower here due to a lack of certain capabilities. For example, it was necessary to create an assembly language routine to do a bit-by-bit logical "AND" of two characters.

8.6.3 Extent of Use

PASCAL/MT is a supported compiler, and has many uses, but the vendor providing it is a very small company. Neither PASCAL is on an automatic update service.

8.6.4 Assembly Language Linkage

PASCAL/MT only provides for use of assembly language as in-line direct code. This is not serious, but it is indicative of a serious lack which was

not scored as part of the FOM analysis. PASCAL/MT does not allow for separate compilation and link of modules. Without this capability, construction of large programs is very difficult.

8.6.5 Readability

PASCAL requires that subordinate procedures precede the procedure or program body that invokes their use. This is inconvenient compared to a language such as "C" which allows specifying top-level procedural modules first and then developing subordinate modules in the source code file.

8.7 PLI-80

PLI-80 is a microprocessor implementation of the subset G of PL/I. It is a new language, but with a large number of uses. The literature written about it tends to indicate a business programming orientation. Its record definitions are very COBOL-like. Variable definitions can become complicated and difficult to understand. It scored neither very badly nor well on most features and its overall score, and its apparent overall suitability for the application, was low.

8.7.1 Data Representation

PLI-80 is a fully block-structured language with a fairly rich set of data types. Numeric data types include: binary integers, binary floating point numbers and BCD (binary coded decimal). Non-numeric types include: bit vectors having lengths of from one to sixteen bits, pointer variables, character strings and records. No provision exists for pointer arithmetic; the user must resort to "tricks" in order to accomplish any pointer arithmetic. For example, in order to increment the pointer variable p, it is necessary to do the following:

```
unspec(i) = unspec(p)
i = i+1
unspec(p) = unspec(i)
```

where i is a temporary fixed binary variable. PLI-80 records may be nested. Storage sharing, representing a single area of memory as consisting of

different data types at different times is not as convenient as with "C" or "PASCAL" (or even FORTRAN). The PLI-80 user must declare based variables (i.e., variables based upon pointers) representing the alternate variables and then set a pointer to point to the desired area of memory. This is both cumbersome and inefficient. PLI-80 allows both static and automatic variables, but treats all variables as static variables. Recursion is achieved by placing all local variables on the stack upon entering a procedure and subsequently restoring these variables from the stack upon leaving the procedure. This does not provide fully reentrant code.

8.7.2 Systems Programming

PLI-80 bit vectors may be used to perform bit-wise logical operations. Hbwever, bit manipulation facilities are inferior to those provided by "C." PLI-80 provides a number of intrinsic functions for string and bit vector manipulation, type conversions, etc. These intrinsic functions are, in general, designed for business applications programming and do not provide the flexibility of the standard "C" library functions.

8.7.3 Control Structures

PLI-80 has most of the modern control structures. There is no "repeat until" or "case" construct. PLI-80 provides a "computed GOTO" very much like the FORTRAN "computed GOTO."

8.7.4 Readability

PLI-80 programs are fairly readable. PLI-80 does not require that functions and procedures be declared before being invoked, thus allowing the programmer more freedom in the organization of a program. The lack of certain control constructs (break, repeat until, case) leads to the use of the "GOTO" statement to a greater extent than in PASCAL or "C."

8.8 PLMX

PLMX is an implementation, targeting a variety of processors, of the Intel PL/M. It implements most of the features of PL/M except reentrancy.

8.8.1 Data Representation

PLMX has most appropriate data constructs and it is a full block-structured language. PLMX variables are address (16 bit) and byte (8 bit). Definition of constants call for hexidecimal values of ASCII characters. It uses a scheme of implementing pointers rather different than PASCAL. It bases a variable or a structure at an address, then changes the contents of the address to move the "pointer." PLMX does not handle floating point numbers, nor does it handle negative numbers. The benchmark implementation required a special assembly language routine for the arithmetic. It also required printing the negative number as a large positive number and interpreting it in its 2's complement form.

8.8.2 Systems Programming

The PLMX strong point is systems programming, and it should have scored better than it did.

8.8.3. Control Structures

PLMX does not have a case statement.

8.8.4 Extent of Use

PL/M was one of the first HOLs for microprocessors. It is extensively used for control applications. PLMX is a new, but professionally supported version with few users.

8.9 PLZ

PLZ is a Zilog family of languages which currently consists of PLZ/SYS and PLZ/ASM. PLZ/SYS is the high order language in which the benchmark was coded, and PLZ/ASM is a structured assembler. Zilog considers PLZ/SYS as a systems programming language. It is not a general-purpose language and it is expected that most programs written in PLZ/SYS will be modest in size. PLZ/SYS is very PASCAL-like in some ways, especially in allowing type definition and having strong type checking. The structure constructs are "IF-ELSE-FI", "DO-OD" (the only loop construct), and "CASE" as part of an extended "IF" including an "ELSE" clause. The scores should likely have been higher in Data Representation, but lower in Systems Programming since most bit-level actions must be performed by assembly language routines. However, a weakness in data representation is a (poorly documented) problem in the use of negative numbers in arithmetic division. The PSE is limited to that provided by Zilog and is not, especially the rudimentary editor, an effective tool. The compiler diagnostics were excellent. One has the option of compiling to machine code or to an intermediate Z-code which is executed interpretively.

8.10 Results

Although some of the individual scores might have properly been higher or lower, it generally appears that the overall results indicate the capability of each language to support the DCT application. Appendix E gives the numerical results and graphically shows the relative scores.

8.10.1 FOM Without Benchmark Results

Four languages, the PASCALs and the "C"s are grouped at the top, with FORTRAN a poor last. See Figure 8-1. These scores reflect the evaluator's feelings about the suitability of the languages, aside from efficiency considerations.

8.10.2 Final Figure of Merit

The top 4 languages change relative positions when benchmark results are included. See Figure 8-2. The strong benchmark showing of Interactive C, combined with the wide variability of the results, puts Interactive C at the top of the list. FORTRAN shows even more poorly due to its time and space inefficiency.

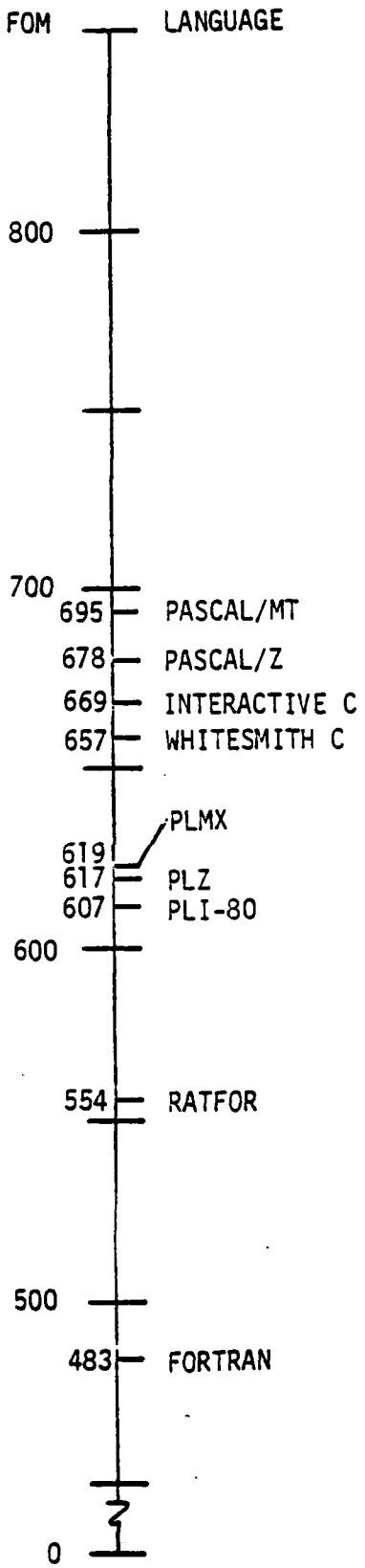

Figure 8-1. Figures of Merit without Benchmark Values

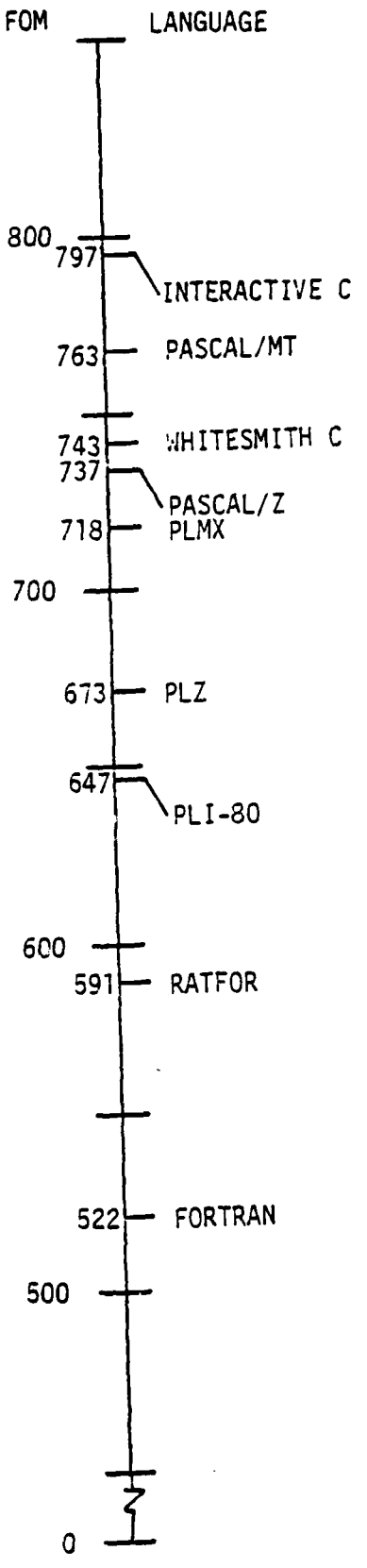

Figure 8-2. Final Figures of Merit

SECTION 9. INTEGRATED ANALYSIS

This section describes the process that integrated the benchmark and FOM analysis.

9.1 General

The methodology of the study calls for a final, ultimately subjective, evaluation and determination of the language to be specified for DCT programming. The benchmark results and the FOM are only a general guide. The FOM figures, especially the FOM without the benchmark values, tend to point to a group of four languages: the two versions of "C" and the two versions of PASCAL. These same languages were felt by the principal investigators to be the best of the final 9 considered. PLMX might well also have been included, but the lack of capability to handle negative numbers was too large an issue to overcome. Either the compiler would have to be modifed or a group of special purpose utility procedures would have to be prepared and used each time numbers were manipulated or compared. The analysis below first compares the versions of "C" and PASCAL to each other, then compares the preferred version of "C" to the preferred version of PASCAL.

9.2 PASCAL and "C"

This paragraph summarizes the analysis which led to determination of the preferred version of the languages PASCAL and "C".

9.2.1 Two "C" Languages

Of the two versions of "C," Interactive C and Whitesmith C, the Interactive C scored higher on the FOM. It is better in space efficiency, provides better access to assembly language, and has a much superior PSE. Interactive C is preferred.

9.2.2 Two PASCAL Languages

Of the two versions of PASCAL, PASCAL/MT and PASCAL/Z, the PASCAL/MT scored higher on the FOM. It is faster, but requires more memory than the PASCAL/Z. The PASCAL/MT however, does not allow the linkage of separately compiled modules and does not provide intermediate assembly language code. It provided very meager diagnostics on the error-seeded version of the benchmark. For these and various other reasons, including consideration of the code in the run-time library, the consensus was to prefer PASCAL/Z.

9.3 Language Selection

The final decision was based upon a group discussion and consideration of the two languages' strengths and weaknesses as discussed in Section 8. In the end, the strengths of the PASCAL were considered to be its increasing use as a teaching language, and its similarity to Ada, especially in the ability to define new data types and to perform extensive compile-time type checking. It was considered to be more readable than "C." Interactive C's strongest points were its strength as a systems programming language and its efficiency. Since it is implemented by a cross-compiler on a PDP-11, it has superior ability to be used effectively in a sophisticated program support environment. "C" also allows considerable flexibility in use of expressions which mix the types of character and integer. Both languages were weak ir the area of compiler support from their respective vendors. After considerable discussion of these topics, and of the likely size and volatility of the DCT programs, the general consensus was that the preferred language for DCT programming was Interactive Systems C.

9.4 Other Alternatives

If for some reason Interactive C could not be used, there are other feasible choices. The Whitesmith C and PASCAL/Z would be suitable choices. The Whitesmith C implemented as a cross-compiler could also be considered. The PASCAL/MT could be effective, but only if the soon-to-be-released upgrade were to perform as advertised and if the company providing the language were to become more broadly based. PLMX has a number of attractive features, but

its use should be predicated on compiler modifications to allow negative numbers and reentrant code. None of the other languages are suitable for large programming projects for a microprocessor-based real-time system.

9.5 DoD Inst 5000.31

Unfortunately, none of the effective languages are listed in DoD Inst 5000.31 as approved languages. The only one of the final candidates which is an approved language is FORTRAN. Even then, Microsoft FORTRAN-80 does not adhere to ANSI-66 or ANSI-77 standards. The only reason that FORTRAN was allowed to proceed to the final evaluation was that it was on the approved list. FORTRAN is not a systems language. Its weaknesses were described in Section 8. If an attempt were made to program the DCT in FORTRAN, it is conservative to estimate that each production model would have only one-half the capability it would have if it were programmed in "C." The development cost for the less capable device could well be two to three times as high. The life cycle support considerations could be even worse. The programs would not be easily modifiable or extendable. Desirable enhancements required for FMF use would be delayed and the resulting code would be expensive to maintain. Based upon the quantitative benchmark results, the figure of merit analysis, and the consensus that it would result in costly development and reduced DCT capability, FORTRAN is judged to be unsuitable for DCT use from both a technical and cost-effectiveness point of view.

9.6 Program Support Environment (PSE)

This paragraph discusses the importance of the PSE, and the resulting requirement for a sophisticated PSE consisting of a minicomputer software engineering host, and a microcomputer development system (MDS).

9.6.1 Importance of the PSE

An important element of consideration in language selection is that of the Program Support Environment. The PSE is the environment in which the DCT software will be developed and maintained. The proper PSE contributes significantly, out of proportion to its weighting on the Phase I Delphi, to

the three identified operational characteristics of Development, Effectiveness, and Life Cycle Maintenance. These aspects are addressed below.

9.6.1.1 Development

The PSE need not be very extensive for small programs of 8-15K bytes with one or two programmers. The programmers can maintain adequate control over the various development versions, alternate approaches, and necessary test harnesses and simulations. For large programming systems however, a more formal method of building libraries, accounting for source code versions, and general configuration control is required. The advantages of such PSE capabilities are faster and easier program development and, ultimately, lower cost.

9.6.1.2 Effectiveness

The proper PSE can contribute to operational effectiveness by enhancing the correctness of the programs developed. It does this by providing test tools that are easy and fast to use, and by providing control over old tests and test results to support regression testing. Fast compiles on the host machine encourage good testing at the programmer level. These factors contribute to both development and maintenance of the software.

9.6.1.3 Life Cycle Maintenance

The same factors that contribute to success in the areas of development and effectiveness are important for Life Cycle Maintenance. The likely volatility of the DCT programs, and the likely numerous different user communities, makes configuration control over versions, variations, and releases even more important than it might otherwise be.

9.6.2 PSE Requirement

For effective development of software for large software projects involving many programmers over a long period of time, a PSE is required that allows the programmers common access to files, provides fast response to

commands, and includes a system for control of the code to be developed. Such an environment is sometimes called a "programmer's work bench," or "software factory." At the same time, for microprocessor oriented projects, the PSE must include a more hardware-oriented capability to allow test and integration of the hardware and software. The solution is to have a PSE that includes several levels of capability. The software development, source code control, and compilation, should be done on a minicomputer software engineering host. The development may also include initial testing of the logic for some modules of code. The programs are then down-loaded to an MDS for final testing and integration with the hardware. Of course the final level, connected to the MDS, is the target NSC800 in a DCT. The general capabilities of the minicomputer and MDS are described below.

9.6.2.1 Minicomputer

The minicomputer software engineering host should first of all include the basic capabilities of a good operating system, based on large fast disks, to allow effective programmer communication. There should be a good mail system, editor, librarian, directory, etc. There should also be a compiler to target the minicomputer host and a cross-compiler to target the NSC800. There should be a system for controlling source code, keeping track of versions, and requiring reasons to be given for changes. A method should be provided for automatically constructing new versions of developing systems. The automated method should keep track of changes to the extent that it recompiles and links only necessary modules (and does include all necessary modules) which have had changes to them. Such a system is useful for small projects, mandatory for large ones. The Interactive Systems PWB/UNIX, with INED editor, Source Code Control System (SCCS), the "Make" facility, and ZCC cross-compiler, satisfies all the stated requirements.

9.6.2.2 MDS

The microcomputer development system should first of all include an effective operating system (O/S). CP/M, provided by Digital Research, is the most common microcomputer system O/S and would be an appropriate choice. It is described in the paragraphs to follow. A recent Mini-Micro Systems survey

on development systems, reference (1), covers typical capabilities and typical systems. The MDS should include a means for loading programs in memory and an effective debugging tool to include multiple breakpoints, trace, disassembler, assembly of single instructions into memory, and ability to display memory and listings at the same time. It should also include a logic analyzer or equivalent means of monitering bus activity, CPU signals, and other activity in the target device. It should include in-circuit-emulation, preferably with the capability to migrate functions step-by-step to the target system. A number of low cost systems (less than $30K) would satisfy these requirements, with a typical system being the Hughes H800 MDS.

9.6.2.2.1 CP/M Definition

CP/M is a monitor control program for microcomputer system development. The basic system includes the monitor, an editor, assembler, link/loader, and several utilities including Peripheral Interchange Program (PIP) and Dynamic Debugging Tool (DDT), a very powerful symbolic debugger. The monitor provides rapid access to programs through a comprehensive file management system, supporting a named file structure and dynamic allocation of file space as well as sequential and random file access. CP/M supports most of the high order languages available for microprocessors, facilitating compatibility and transportability among different microprocessor systems.

9.6.2.2.2 Additional CP/M Capabilities

In addition to the editor provided with CP/M, there are several powerful screen editors available for CP/M systems, such as WORDSTAR and WORDMASTER. These editors are equal to or better than most available on large minicomputer systems. Both offer easy and efficient methods of entering and altering source code files. One of the outstanding subsystems of CP/M is DDT, which allows on-line debugging of executable programs. DDT provides a symbolic assembler/disassembler, a single or multiple step trace feature with automatic register/flag and instruction display, breakpoints, register and memory display and alteration capability, and a memory block move function.

9.6.2.3 Target System

Lastly, the development facility must include an actual DCT with the embedded NSC800 microprocessor.

9.7 Risks

Interactive C is well suited to the DCT application. The nature of the language itself, the efficiency of the compiler, and the capability of the program support environment are all such that there is little risk for program development. The only risk is in the area of compiler support. Interactive Systems does not provide the compiler as a supported product. However, Interactive Systems does provide the source, written in "C," for the compiler and it is now supported at Naval Ocean Systems Center (NOSC), San Diego. In summary, the risks involved with the selection of Interactive Systems C appear to be minimal, and no greater than those which would be associated with the selection of one of the other desirable HOLs.

9.8 Operational Effectiveness

There appear to be no issues important from the viewpoint of operational effectiveness, man-machine interface, communications processing, data storage and manipulation, etc., that are not handled well by Interactive Systems C. Use of the language will allow the development of software that will be operationally effective.

SECTION 10. CONCLUSIONS

This section presents the study conclusions and additional information about the language selected.

10.1 The Language

The language selected for DCT software development is the version of "C" provided by Interactive Systems, Santa Monica, California. "C" is a modern general-purpose programming language, originally developed in the early 1970's at Bell Labs. Interactive Systems provides a cross-compiler hosted on a PDP-11, and targeting the Z80. Additional information is provided below.

10.1.1 General

"C" is not a "very" high level language. Although it has the desirable characteristics of a HOL, it does not provide facilities such as operations which deal with composite objects, e.g., structures, arrays, and strings, as a whole. It does not provide for multiprogramming or parallel operations, nor does it inherently provide input-output facilities or file access methods. Some of these capabilities are provided by standard library routines, or are not needed for DCT applications. "C" is a relatively low-level language: it deals with objects such as characters, addresses, and integers much as does a real machine. It is a relatively small language, although with many operators, and hence is relatively easy to learn and easy to write compilers for.

10.1.2 Language Features

Specific language features have been described in Section 8. In summary, "C" has been called a systems programming language, but has also been used for numerical, data base, and text processing applications. It provides all necessary means of data representation and the fundamental flow of control constructs necessary for well-structured programs. It is not strongly typed in the sense of PASCAL, nor is it a completely block structured language in the sense of nesting of procedures. It is however, largely block-oriented

with ample control over scope of variable definitions. It has the capability of being concise to the point of being cryptic, and requires adherence to good standards. The Interactive Systems C provides an outstanding program support environment, suitable not only for small programs, but also for large projects, such as the DCT, involving many programmers.

10.2 Relation to Instruction Set Architecture

The "C" language was originally designed with the PDP-11 in mind as a target processor. The Z80, although not as powerful a machine, does have the necessary features, including explicit index registers and addressing modes, to be compatible with the nature of the language. As shown by the benchmark results, especially execution time, the compiler generates efficient code.

10.3 Program Support Environment

The identification of the PSE was related to the selection of a language since it affects how effectively the computer programs can be developed and maintained. A conclusion of the study is that a three level PSE is necessary for effective software development, regardless of language selected. The PSE levels are:

- Minicomputer software engineering host
- Microcomputer Development System
- Target computer (NSC 800) in the DCT

At the present time, the appropriate system for the software engineering host is a PDP-11/70, or other PDP-11, with Interactive Systems software. There are a number of systems suitable as the MDS. A typical system is the Hughes H800 MDS. Before selection of a specific suite of hardware and software, it is necessary to identify the software development and maintenance strategy. The contractor selection and whether or not the software is developed on-site at MCTSSA co[illegible] have a bearing on the PSE. Since a number of specific equipments are suitable for the PSE, the selection of a specific suite of PSE hardware and software should be deferred and addressed as part of the decision regarding the DCT software development and maintenance strategy.

10.4 Summary

The DCT software should be developed in Interactive Systems C in a sophisticated program support environment.


DATE
FILMED
9-8